Peer-to-Peer Prefix Tree for Large Scale Service Discovery

Cédric Tedeschi

Numéro d'ordre : 480 *Numéro attribué par la bibliothèque : 07ENSL0480*

École Normale Supérieure de Lyon

Laboratoire de l'Informatique du Parallélisme

THÈSE

pour obtenir le grade de

Docteur de l'Université de Lyon - École Normale Supérieure de Lyon

spécialité : Informatique

au titre de l'école doctorale de mathématiques et d'informatique fondamentale de Lyon

Présentée et soutenue publiquement le 2 octobre 2008 par

Cédric Tedeschi

Peer-to-Peer Prefix Tree for Large Scale Service Discovery

Directeur de thèse : Frédéric Desprez
Co-encadrant de thèse : Eddy Caron

Après avis de : Luigi Liquori
: Sébastien Tixeuil

Devant la comission d'examen formée de :

Eddy Caron, membre
Ajoy K. Datta, membre
Frédéric Desprez, membre
Luigi Liquori, membre/rapporteur
Pierre Sens, membre
Sébastien Tixeuil, membre/rapporteur

Remerciements

Le train bouge, les réverbères à deux têtes qui séparent le quai s'alignent et s'éloignent, ça doit être signe que c'est fini... Pourtant, j'aimerais faire le voyage en arrière, en accéléré. Je suis assis dans le sens inverse de la marche, suffit peut-être de synchroniser mon esprit... Juste avant, il y a eu la soutenance. J'aimerais dire merci aux membres du jury, Ajoy K. Datta, Luigi Liquori, Pierre Sens et Sébastien Tixeuil d'être venus pour ça, pour leurs questions et leurs retours, pour avoir fait de cette soutenance ce qu'elle a été. Peu de temps avant, je rédigeais, et je souhaiterais remercier les rapporteurs, Luigi et Sébastien, d'avoir accepté la charge supplémentaire de relire cette thèse.

Les dernières lumières de la ville s'enfoncent sous le ciel noir, il reste quelques reflets qui dansent à la surface du fleuve... Pendant trois ans, mes directeurs de thèses, Eddy Caron et Frédéric Desprez ont balisé une route pour éviter que je me perde au milieu de cette jungle. Merci à Eddy pour avoir soutenu, défendu et profondément cru en ce que nous avons fait. Pour m'avoir fait répéter mes slides à l'autre bout du monde, pour avoir vérifié mes algos à la main, même si ça devait prendre la journée. Quoiqu'un doctorant puisse attendre de ses encadrants, il m'a offert bien plus. Merci à Frédéric, pour m'avoir proposé un sujet de stage il y a 5 ans, pour m'avoir poussé à candidater au Master de l'ENS, pour m'avoir sorti la tête du guidon aux bons moments, et pour sa vision de la recherche. Franck Petit est en quelque sorte mon troisième directeur de thèse, je sais que c'est une chance d'avoir pu travailler avec lui. Merci d'avoir marqué ce travail de ta science, et d'avoir toujours répondu patiemment, en détail, à mes questions bizarres sur les systèmes distribués. Enfin, merci à Ajoy, pour avoir eu la patience de m'écouter et de comprendre ce qu'on faisait. Merci pour ces heures passées à essayer de construire un pont entre nos deux communautés.

Au milieu de la nuit, le train passe dans la vallée, l'air est froid, et la lune éclaire les montagnes qui paraissent bleues... Je me souviens que Benjamin et Gaël m'ont aidé à faire en sorte que la soutenance se passe bien, qu'ils ont traversé la rue du Vercors sous la pluie pour tester le vidéoprojecteur quelques heures avant. Un peu plus loin, il y a toujours Ben, des courses de chaises, des arts martiaux avec des tubes en cartons... Merci de m'avoir suivi en moto sur des routes douteuses et humides, jusqu'à cette chapelle qui domine l'ouest lyonnais. Il y a aussi le *Salut le chat band*, merci à Matthieu pour avoir tenté autre chose que mon éternel blues en douze mesures (même si des fois je le tente en mineur... bon ok, ça compte pas), et à Aurélien pour avoir toujours réussi à me suivre malgré mes syncopes plus ou moins volontaires. Merci à Veronika pour m'avoir laissé croire que j'étais fort au squash (surtout en défense). Merci à Raphaël, pour m'avoir aidé deux fois à déménager... ça paraît tellement plus facile quand c'est toi qui range le camion... Merci à Emmanuel, pour ces parties d'échec à Saint-Georges et pour ces nuits à stresser ensemble, au laboratoire, à essayer de savoir s'il y avait une chance qu'on ait ce financement...

Derrière, il y a encore des trucs... Merci à Jean-Sébastien, pour m'avoir laissé sauter à pied joint sur son bureau, avoir répondu une fois sur trois à mes questions, et pour ces nuits à errer le long des quais de Saône, à regarder ce poisson stagner en essayant de remonter le courant sous les lumières des ponts... Le lendemain, il a du trouver le chemin de la mer, pendant que je me perdais dans des vapeurs de café teintées de vanille, jusqu'à ce que le soleil d'hiver s'enfonce et que je regarde une dernière fois la fontaine et les réverbères oranges s'allumer.

Parfois la famille semble un dernier refuge. Merci à mes parents, et en particulier à mon père qui croit en moi plus que moi-même. À mes grands-parents et à Laurent, gardiens d'un certain équilibre. À mon frère, Florian, pour avoir été là, entre Nîmes et Perpignan (et ailleurs), et pour ce qu'il est, simplement. Enfin, à Lola, pour me permettre d'être autre chose qu'un fantôme qui a froid, perdu sous le ciel et les nuages.

Contents

Introduction

Solving scientific problems requires more and more computational resources. The amount of data to be computed and the complexity of simulations in areas such as bioinformatics, high energy physics, or cosmology, are still growing. To reduce the time needed for such computations, a first approach consisted in using parallel machines, also known as *supercomputers*, composed of hundreds, or even thousands of identical processors connected by high bandwidth I/O systems. However, this type of infrastructures is very costly and scientists do not always have the means to acquire a supercomputer.

With the emergence of high performance networks, a cheaper alternative has consisted in connecting standard computers through a network. When the computers are located in the same place like a university or a company, we use the term *cluster*. If geographically distributed, these infrastructures are called *computational grids*.

Problems to Be Solved

Interactions of ocean and atmosphere at the Earth's surface and frictional forces at work where the air and water interface, like the "*El Niño*" phenomenon, can result in ecological drastic changes and human disasters. Understanding the underlying mechanisms of climate changes produces a tremendous amount of modeling and simulation requiring a proportional amount of computing power [93].

Setting the parameters of a nuclear reactor contains a lot of dangers and, as every one knows, can result in great disasters if not properly done. That's why numerical simulation is required beforehand. As explained by Donald B. Batchelor in [27]: "*In a magnetic fusion device, [the] plasma is maintained in a largely self-organized state that far from equilibrium the mathematical description of which is characterized by high dimensionality, nonlinearity, extreme range of time and space scales, and sensitivity to geometric details. High-performance computing plays an essential role in fusion research not just to understand the theory and make quantitative comparison to experiments, but also to provide direct support to the experiments by interpreting measurements and designing experiments.*"

Understanding the formation of the structures (galaxies) of the universe is one great challenge for which astrophysicists are today ready to engage a lot of energy. Among the means for solving this issue, simulation seems today the most important. As explained by people of the Horizon Project [144], the goal is, neither more nor less, the simulation of the formation of the universe. As you can read on the site of the project, they have made "*the largest N-body simulation ever performed*". As a matter of fact, they have simulated a 13.7 Gigayear long evolution of 4096^3 dark matter particles. This experiment, that would have taken more than a thousand years on a single computer, required running the simulation code for a couple of month on the 6144 processors of the new BULL supercomputer of the CEA (French atomic energy commission) supercomputing center.

The structure of biomolecular machines is the foundation of living systems. The difficulty for

observing the underlying mechanisms of life at this scale and the need for molecular dynamics simulations at this scale is well highlighted by Schulten *et al.* in [127]: "*Genetic sequences stored as DNA translate into chains of amino acids that fold spontaneously into proteins that catalyze chains of reactions in the delicate balance of activity in living cells. Interactions with water, ions, and ligands enable and disable functions with the twist of a helix or rotation of a side chain. The fine machinery of life at the molecular scale is observed clearly only when frozen in crystals, leaving the exact mechanisms in doubt. One can, however, employ molecular dynamics simulations to reveal the molecular dance of life in full detail.*" For a few years now, bioinformatics has become one of the most common applications for grid computing.

The Grid Purpose and the Reality

Primary goals of Grid Computing are basically to connect any available computing resource (supercomputers, clusters of processors, desktop computers) at any place in the world and aggregate them into a unique virtual entity called the *grid*. The ultimate purpose of Grid Computing, as expressed by Foster and Kesselman at the very end of the twentieth century [62], is to offer this computing power to anyone (scientists, but also end user at the edge of the Internet) plugged to the network in a transparent way, just as easily as switching the light on after having plugged the jack into the socket.

This ease and transparency are today far from being a reality. Many barriers hinder such a transparency. One of the main concepts behind them is *heterogeneity*, both in terms of hardware (computing power, network accessibility, or storage space) and software (operating system, communication protocols), making the problem quite delicate. Among the numerous other problems making computational grid quite hard to maintain and use, we find the security issues, such platforms not being spared by attacks, the scheduling, whose goal is to efficiently schedule computing jobs to avoid starvation or unfairness in the distribution of resources to still growing number of users, the fault-tolerance, whose goal is to keep the platform efficient when crash failures occur, the scalability, whose goal is to allow the system to keep working even when highly distributed and when the number of services and requests drastically increases.

Another fundamental issue, a preliminary process of any computing platform, is the connection to be made between a user's need (expressed through a request) and the different computing abilities available on the platform, gathered under the term *services*.

Problematic

Service Discovery

A Basic Example. Any device in a computational grid, provide some computing abilities. They are for instance able to multiply matrices. This ability, if offered to the community, can be called a *service*. Now imagine that somewhere in the world, a user — application or human, needs to multiply two matrices but is not able to do it, for instance because he/she/it does not have any matrix multiplication program locally. As one can imagine, it would be interesting for the user to use the service in a remote mode. But before to be able to do so, the user needs to discover the service and learn how to connect to the service (protocol, address, ports) and how to use it (encoding of the matrices, location of the result). What is missing is a directory of services the user could consult to find what he/she/it needs.

Service-Oriented Architecture (SOA). This notion of service was generalized by the SOA reference model [3], which is an attempt to define a standard architecture for using business computing entities. SOA is introduced as a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains. The SOA standard describes a service as a mechanism to enable access to one or more capabilities, where the service is accessed using a prescribed interface and exercised consistent with constraints and policies as specified by the service description. This description is what our user is looking for.

Emerging Platforms

Our problematic is to make possible such a service discovery in platforms emerging today which are large (gathering a high number of geographically distributed nodes), heterogeneous (in terms of hardware, operating systems, and network performance), and dynamic (processors are constantly joining and leaving the system without notice). Moreover, in such large environments, the number of services and requests are also constantly growing. Under these conditions, maintaining a view of the available services and answering the requests becomes a far more challenging problem than creating a simple directory. This is this challenge that we address in this dissertation.

The Contribution of the Peer-to-Peer Community

As we said, new infrastructures hosting computational grids become larger and more unstable. As a consequence, they fail to provide a minimum stable set of components that could be able to maintain a global view of the platform. Grid software, initially designed assuming such a stable infrastructure needs to be redesigned, to be able to face the nature of new platforms. Software introduced by the peer-to-peer community, offering purely decentralized techniques to share resources on highly large and dynamic platforms has appeared to be of high interest for the grid community. Peer-to-peer technologies offer more and more robust tools for large scale content distribution (content being possibly information on available services), while addressing failures (leaving processors). As a consequence, it has much to offer to the grid computing community, that addresses infrastructure, but not the scale and the failures. This convergence were initially put in words by Iamnitichi and Foster [83]. Our solution is an example of this convergence.

The Contribution of the Self-Stabilization Community

On platforms where processors are constantly undergoing departures or failures of processors, one requirement is to be able to remain efficient, whatever the number of departures and/or crashes is. *Self-Stabilization* is a general powerful technique to design a system tolerating arbitrary transient faults. In other words, regardless of the number of crashes, failures (in one word, the arbitrary *state* of the system), the system will be able to rebuild a consistent one. Our solution calls upon self-stabilization for fault-tolerance.

Dissertation Organization

In Chapter 1, the information required to understand the issue tackled in this dissertation, and how this problem has been approached on early grid computing platforms, is detailed. A concise statement of the problem is provided, along with an overview of our solution and a detailed outline.

In Chapter 2, we provide the background and related works in peer-to-peer systems and self-stabilization allowing to understand our solution and compare it with similar works, on the relevant points (complexities, scale, efficiency according to several parameters, load balancing, fault-tolerance).

Chapters 3, 4, and 5 present the theoretical part of our contribution. In Chapter 3, the basic design of our system is presented. Our solution is a two-layer peer-to-peer architecture. In this first *contribution chapter*, we focus on the upper layer which is a logical prefix tree indexing the information and built as some services join and leave the system. In Chapter 4, the lower layer of our architecture is studied. The problem of efficiently mapping the upper logical layer indexing the information on services, onto the lower physical one, representing the network is addressed. Finally, our theoretical contributions ends with Chapter 5 where the fault-tolerance issue within our architecture is addressed. Our solution strongly relies on the self-stabilization paradigm.

Chapter 6 presents the practical aspects of our contribution, namely (1) a preliminary peer-to-peer extension of an existing grid software, and (2) the prototype implementation of our solution, its early experimentation and its application to an important emerging problem: network reservation and provisioning.

Chapter 1

Preliminaries

In this chapter, we give the information required to understand the problem tackled in this dissertation and its context, namely *Service Discovery in Grid Computing*. In the next section, we briefly recall the genesis of Grid Computing and the two main types of infrastructures it traditionally considers. We illustrate the tremendous amount of efforts in grid software development by some important examples of leading grid middleware projects. In Section 1.2, we present the purpose of *Service Discovery*, and illustrates it with few relevant traditional approaches for designing and implementing a service discovery system. In Section 1.3, we give a statement of the problem we are dealing with. Finally, in Section 1.4, we introduce the remainder of the dissertation by giving an overview of the solution we have developed.

1.1 Grid Computing

In many areas, such as cosmology, bioinformatics, or high energy physics, new problems involve a lot of simulations and modelling, requiring an exponentially increasing computing power. Local aggregations of processors, called *clusters*, appear inadequate to solve these problems. At the very end of the twentieth century, this need for a greater computing power, along with the explosion of distributed computing resources connected by high-speed networks gave birth to a new field of computer science, known as *Grid Computing*.

As early defined by Foster and Kesselman [62], the ultimate purpose of *grid computing* is to offer an aggregation of these worldwide distributed connected computing resources to the scientific community, governmental institutions or users at the edge of the Internet. As achieved with the electrical power grid, one hundred years earlier, this should be done in a transparent way. Nonetheless, due to several factors, this aggregation is a very challenging issue. Heterogeneity — or even incompatibility, both in terms of hardware (computing power, network accessibility, or storage space) and software (operating system, communication protocols) makes this transparency problem quite delicate.

Within the grid computing area, we can distinct two main types of infrastructures, which appeared in parallel, most of the time referred to as *Remote Cluster Computing* and *Global Computing*. Software tools aiming at facilitating the exploitation of such infrastructures — aggregating the computing power and making it available while hiding heterogeneity and volatility, are known under the general term *Grid Middleware*. We now review important existing infrastructures along with widely used middleware for both types of grids.

1.1.1 Remote Cluster Computing.

Infrastructures. The *Remote Cluster Computing* paradigm aims at using the computing power of federations of geographically distributed computer clusters of private, governmental or academic institutions. The physical resources are virtualized and grouped within logical *Virtual Organizations* (VOs). The purpose of this virtualization is to provide to users an access point to these resources without requiring any modifications in their programs. These grids offer a good level of availability and stability since most of the time dedicated to computing.

TeraGrid. TeraGrid [138] is a good example of such grids. It is an open infrastructure combining leadership class resources at eleven partner sites to create an integrated, persistent computational resource for scientific computing. TeraGrid includes more than 250 teraflops of computing capability and more than 30 petabytes (quadrillions of bytes) of online and archival data storage with rapid access and retrieval over high-performance networks.

LCG. CERN (European Organization for Nuclear Research) has recently chosen grid technology to solve a huge data storage and analysis challenge brought by the LHC (*Large Hadron Collider*) [4], the world's largest and highest-energy particle accelerator, which will produce a goldmine for finding traces of new fundamental particles of matter, which in turn will tell physicists a lot more about how the Universe was formed and what its future might be. The data production will be about 15 petabytes a year. The LHC Computing Grid project (LCG) [2], which was launched in 2002, has a mission to integrate thousands of computers worldwide into a global computing resource, which will be used to store and analyze the huge amounts of data produced by the LHC.

EGEE. Launched in 2004 and funded by the European Commission, the EGEE project aimed to build on recent advances in grid technology and develop a service grid infrastructure which is available to scientists 24 hours-a-day. The project was primarily concentrated on three core areas [141]: (1) Build a consistent, robust and secure grid network that will attract additional computing resources. (2) Continuously improve and maintain the middleware in order to deliver a reliable service to users. (3) Attract new users from industry as well as science and ensure they receive the high standard of training and support they need. The EGEE project initially focused on two well-defined application areas: High Energy Physics and Biomedical. Today, EGEE is the largest multi-disciplinary grid infrastructure in the world, which brings together more than 120 organizations to produce a reliable and scalable computing resource available to the European and global research community. At present, it consists of 250 sites in 48 countries and more than 68,000 CPUs available to some 8,000 users.

Grid'5000. Grid'5000 [33] is a french research effort to develop a large scale nation wide infrastructure for Grid research. 17 laboratories are involved, nation wide. As large scale distributed systems such as grids are difficult to study from theoretical models and simulators only, Grid'5000 aims at providing the community of grid researchers a testbed allowing experiments in all the software layers between the network protocols up to the applications. Grid'5000 fills the need for real large scale research grid and is currently one of the most advanced research grid. The current plans are to assemble a physical platform featuring 9 local platform (at least one cluster per site), each with 100 to a thousand PCs, connected by the Renater Education and Research Network [149]. All clusters will be connected to Renater with a 10Gb/s link (or at least 1 Gb/s, when 10Gb/s is not available yet). The interconnection and interoperation of Grid'5000

with other academic grid research infrastructures like DAS-3 [140] in The Netherlands, or Naregi [143] in Japan are currently under study.

Middleware. We now briefly review two middleware systems to build such grids. They are among the most important in terms of number of people involved and number of users and other middleware projects using it as a building block.

Globus. Globus [61] is an open source project providing a set of tools intended to ease the construction and use of grids built on the model of virtual organizations. The Globus toolkit includes software (software services and libraries) for security, information infrastructure, resource management, data management, communication, fault detection, and portability. It is packaged as a set of components that can be used either independently or together to develop applications. Since several years now, Globus has become a widely used solution to build large computing systems. Today, as it intends to provide standard protocols and APIs for grid computing, Globus also serves as a solid and common platform for implementing higher-level middleware and programming tools, ensuring interoperability amongst such high level components.

gLite. Born from the collaborative efforts of more than 80 people in 12 different academic and industrial research centers as part of the EGEE Project, gLite [142] provides a framework for building grid applications tapping into the power of distributed computing and storage resources across the Internet. The gLite grid services follow a Service Oriented Architecture (SOA), meaning that it is intended to easily connect to other grid services. It currently aims at facilitating compliance with upcoming grid services standards, for instance the Web Service Resource Framework (WSRF) [151] from the OASIS consortium [147], a consortium working for the convergence and adoption of standards in information technologies, or the Open Grid Service Architecture (OGSA) [148] from the Open Grid Forum community [5], which gathers a community of grid computing users and developers whose common purpose is the standardization of grid computing tools.

1.1.2 Global Computing.

Infrastructures. The *Global computing*, or *Desktop Computing*, based on *cycle stealing*, aims at collecting computing power of worldwide distributed desktop workstations connected to the web, when unused by their owner. As the performance of desktop computers increases and the number of volunteer grows, the amount of computing power collected allows to solve very large problems. Main issues to deal with when building such platforms are related to heterogeneity and volatility. Public-resource computing emerged in the mid-1990s with two projects, GIMPS and Distributed.net. In 1999, SETI@home [15] was launched, which has attracted millions of volunteer worldwide to help the SETI (*Search for Extra-Terrestrial Intelligence*) project.

Middleware.

Condor. The pioneering system Condor [139] is basically a user-friendly job queueing mechanism for global computing infrastructures. Users submit their serial or parallel jobs to Condor which chooses when and where to run them based upon a policy and monitors their progress until informing the user upon completion. Condor offers a set of techniques to harness wasted CPU power from idle desktop workstations. For instance, Condor can be configured to only

use desktop machines where the keyboard and mouse are idle. Moreover, Condor provides mechanisms to detect activity on workstations and may be able to produce a checkpoint and migrate a job to a different workstation detected as idle.

BOINC. BOINC [14] (*Berkeley Open Infrastructure for Network Computing*), is a software suite which generalizes the SETI@home approach to any public resource computing projects. Numerous *@home* are emerging these days, using the BOINC software. These projects intend to solve problems of different fields of science, like mathematics (ABC@home [7]), or medicine (World Community Grid [73]).

Computing devices of a grid are featured with some resources (computing power, storage space, files, computing software). If they are installed and ready to be used by other users, the become a *service*. One key prerequisite to use these resources, independently from the type of infrastructure, is to make these resources *visible* to potential users. This process of matching needs of users with computing capabilities available on a platform is generally referred to as *Service Discovery*.

1.2 Service Discovery

This section deals with the concept of *service discovery*. In a first part, we define the notion of service, and explain what is generally assumed when using the word *service* in a grid computing context. The second part of this section details a bit further the service discovery process and the traditional approaches developed within early grid middleware.

1.2.1 Service

Notion of Service. Very generally a service can be defined as a network-enabled entity that provides some computing capability. That short definition points out two important facts. First, a service is able to communicate with other entities via a network and second, it answers to a need expressed by a potential user, or *client*. In other words, a service is provided by an entity - the service provider, or *server* - for use by others, or *clients*. Note that the eventual consumers of the service may not be known to the service provider and may demonstrate uses of the service beyond the scope originally conceived by the provider. The Service-Oriented Architecture (SOA)[3] Reference Model is an attempt to define a standard architecture for using business computing entities. SOA is introduced as a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains. A SOA can thus be seen as a means to match needs of clients with capabilities provided by servers, or *services*. The SOA standard describes a service as a mechanism to enable access to one or more capabilities, where the access is provided using a prescribed interface and is exercised consistent with constraints and policies as specified by the service description.

Definition. As our context is grid computing, we restrict the notion of *service* to a remote computing resource able to compute a result given a set of parameters (including data to be processed). The client is not interested in pure computing power, but in computing power able to perform some operation. Consider the astrophysicist trying to simulate the past of the universe. He obviously needs computing power, but he mainly needs the software able to provide this simulation. Then, pure computing power or storage space fall out of our definition. We define a service as a software component (scientific libraries, binary code, scientific simulations) provided with some particular

characteristics related to the operating system, hardware architecture, capacity (free CPU power, free memory or storage space) and location of the service (cluster, administrative domain).

Examples. Scientific libraries like linear algebra packages are typical basic computational utilities used in computational grids. BLAS [57], LaPACK [16], and ScaLAPACK [32], or the S3L library [6] developed by SUN are well-known examples. Today's computational grids aim to offer to scientists a platform providing the computational abilities they require. More complex application are found. We can cite as examples any scientific application related to simulation, or analysis and requiring a very large computing power. Sparse Matrix solvers [13, 51] functions are also currently highly studied for availability as a set of services on the grid, like within the GRID-TLSE project [12]. Among the numerous today's scientific applications and tools already available on grids or candidate for *gridification*, we can cite physics tools like Adaptive Mesh Refinements [99], cosmological grid-based applications like Enzo [111] or Ramses [38], climate predictions applications [37, 58], or the well known BLAST [11], which is one of the most gridified tool in bioinformatics [41, 124].

Visibility. Visibility refers to the capacity for those with needs and those with capabilities to be able to see each other. This is typically done by providing descriptions for such aspects as functions and technical requirements, related constraints and policies, and mechanisms for access or response. The descriptions need to be in a form (or can be transformed to a form) in which their syntax and semantics are widely accessible and understandable.

Three Actors. That definition lays the first stones of the relations between the three main types of actors in our architecture, *i.e.,* servers, clients, and service management entities, sometimes referred to as *agents*. These agents are responsible for the mapping between clients' needs and servers' offers. It also emphasizes the need for a service description necessary for clients to find and choose the service that corresponds best to their needs among the pool of offers.

1.2.2 Discovery

As illustrated on Figure 1.1, *Service Discovery* is a process taking a query as a parameter and returning a set of available servers satisfying the requirements enounced in the query, along with the information allowing to use the service in a remote mode (basically the address of the server). As also suggested on Figure 1.1, a service discovery systems provide two fundamental features:

1. **Service Registration.** When a resource or a service joins a network it has to register itself in a registry. This is necessary for the service to be visible. Upon registration a description of the new service is provided and written along with the service in the registry. That description allows the system to match that specific service with a client request. Handled in the same way, a server may decide to unregister one of its service. This part is illustrated on the right part of Figure 1.1 and is most of the time initiated by the servers (except if the system decides to remove a service from the registry in the case the service has an undesired behavior).

2. **Service Location.** Once a service is registered, the system considers it is available and thus if any client issues a request that matches that specific service, it will be added to the list of appropriate services that is to be returned to the client. The information returned to the client contains the location of the service and how to access to it. (See the left part of Figure 1.1.) The Location process is made in three phases. First, the client sends its request to the system.

Then, the system collects the information of the services matching the request. Finally, the information found is sent back to the client.

Several tools within several computational platforms paradigm tackles the service discovery issue. To give a better illustration of this problem, we now briefly review two widely used approaches for service discovery, namely MDS2, the *Grid Information System* of the Globus toolkit version 2, and UDDI, used in the Web Services technology.

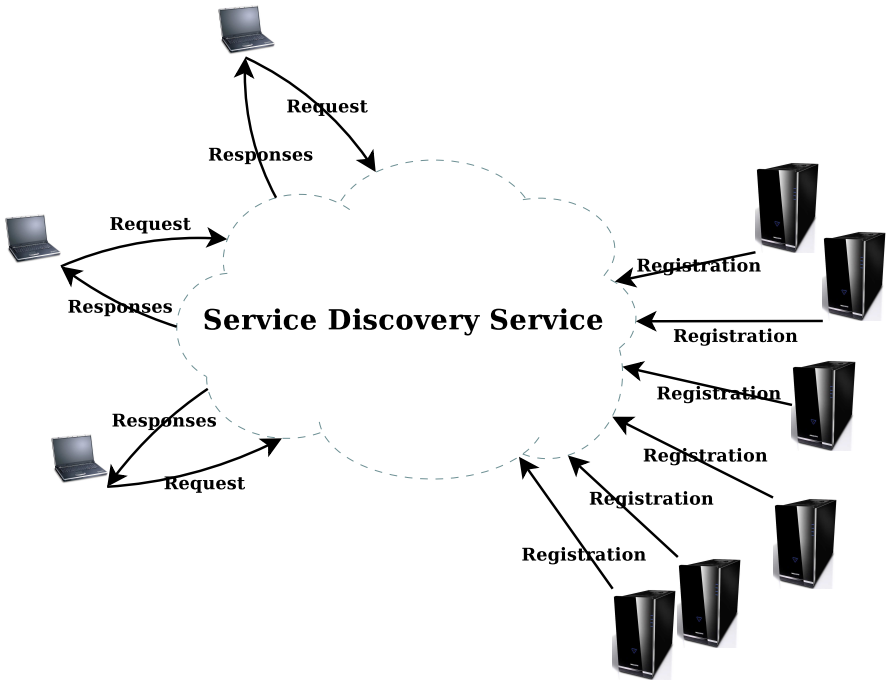


Figure 1.1: Service discovery conceptual architecture.

MDS2

Overview. MDS2 is the Grid Information System of Globus Toolkit version 2. An MDS4 version based on Web Services (refer to next section) technology has been integrated in the Globus Toolkit 4. However, the differences between the versions are not related to the core design itself [86]. This grid information service architecture (see Figure 1.2) comprises two fundamental entities: particular *information providers* and specialized *aggregate directory* services.

- An **information provider (IP)** is a software device providing information on a particular resource (designated by the provider of the resource). Together, information providers form a common infrastructure providing access to detailed, dynamic information about grid entities, independently from Virtual Organizations. (A Virtual Organization can be seen as a set of computation abilities grouped to be used by a particular group of people sharing a particular goal.) For example, a provider for a computing resource might provide information about the number of nodes, amount of memory, operating system version number and load average; a provider for a running application might provide information about its configuration and

current status. In other words, an information provider gives the *raw* information about a resource.

- **aggregate directories (AD)** provide often specialized, VO-specific views of federated resources or services, each VO having its own scope. For example, a directory, intended to support application monitoring, might keep track of running applications. Another directory serving the scope of one VO whose scope is to optimize a particular parameter will provide an aggregate directory on which resources will be sorted according to this parameter.

GRPP. Each IP informs ADs of the availability of the information it provides a resource using the GRid Registration Protocol (GRRP). In other words, the GRRP protocol is used to update information contained in AD. GRRP defines a notification mechanism that one server can use to "push" simple information about the existence of a service to another element of the information services architecture. It is a soft-state protocol, meaning in our context that state established at a remote location by a notification (e.g., an index entry for an information provider) may eventually be discarded unless refreshed by a stream of subsequent notifications.

GRIP and LDAP. Getting information from IPs is done through the GRid Information Protocol (GRIP), which relies on the LDAP [80] protocol. LDAP defines a data model, query language, and wire protocol. Aspects of the data model are illustrated in Figure 1.3. Within information providers, information is structured according to the LDAP model. GRIP is used by client to access informations contained in both AD and IP.

Scenario. When a client needs to discover a resource, it first sends a *discovery* request to a VO-specific AD. This AD serves a given scope and references and presents resources according to this scope. Once the client has received the reference of the service matching his request (and information on how to contact the corresponding IP), it sends a new *lookup* request using the GRIP protocol to retrieve the detailed information of the resource. See Figure 1.2.

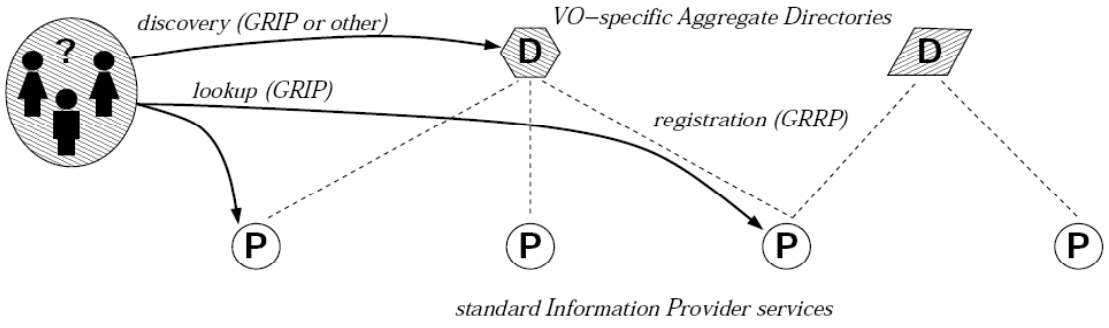


Figure 1.2: MDS architecture overview. Using the GRid Information Protocol (GRIP), users can query aggregate directory services (denoted **D**) to discover relevant entities, and/or query information providers (**P**) to obtain information about individual entities. Description services are normally hosted by a grid entity directly, or by a front-end gateway serving the entity.

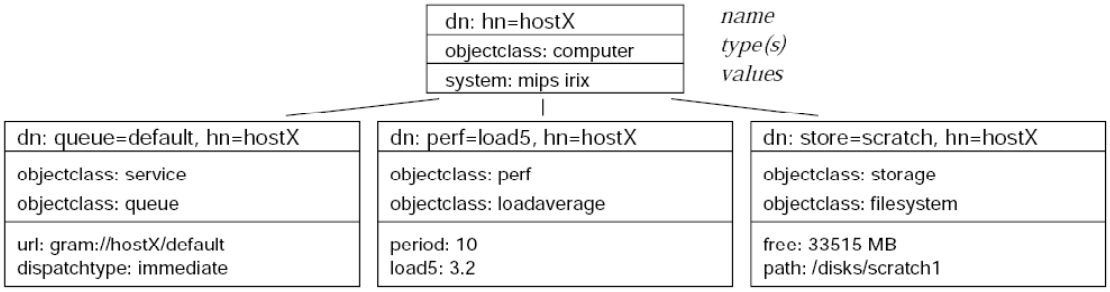


Figure 1.3: The LDAP data model represents information as a set of objects organized in a hierarchical namespace. Each object is tagged with one or more named types. Each object contains bindings of values to named attributes according to the object type(s).

UDDI, Web Services

Web Services Overview. Web Services, whose general architecture is provided by Figure 1.4 is a system designed to support interoperable machine-to-machine interaction over a network [10]. Web services are frequently just Web APIs, that can be accessed over a network, such as the Internet, and executed on a remote system hosting the requested services. Clients and servers (often businesses providing on-line services) communicate using XML messages that follow the SOAP standard. When one want to make its services available through the web, the information is sent to a set of web-based registries, called *Universal Description, Discovery and Integration* (UDDI) in which information on web-services are stored. Clients, for discovery, and servers, for registration of their services, communicate with UDDI registries using the Web Services Description Language (WSDL), a standard XML-based language for describing such services.

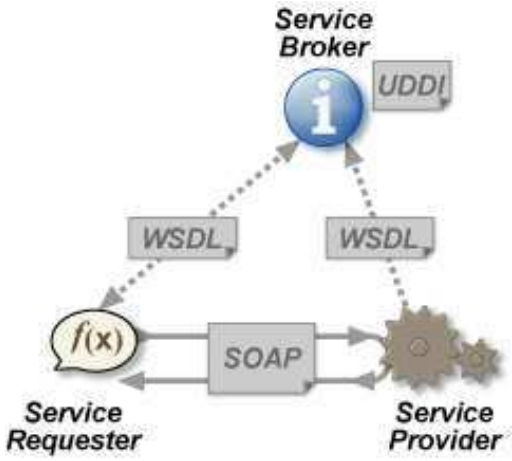


Figure 1.4: Web Services architecture.

UDDI Overview. UDDI is a group of Web-based registries that expose information about a business or other entity and its technical interfaces (or API's). These registries are run by multiple operator sites, and can be used by anyone who wants to make information available about one or more businesses or entities, as well as anyone that wants to find that information. By accessing any of the public UDDI Operator Sites, anyone can search for information about Web Services that are made available by or on behalf of a business. The benefit of having access to this information is to provide a mechanism that allows others to discover what technical programming interfaces are provided for interacting with a business for such purposes as electronic commerce. The benefit to the individual business is increased exposure in an electronic commerce enabled world. The information that a business can register includes several kinds of simple data that help others determine the answers to the questions "who, what, where and how". Simple information about a business, information such as name, business identifiers and contact information answers the question "Who?". The question "What?" involves classification information that includes industry codes and product classifications, as well as descriptive information about the services that the business makes available. Answering the question "Where?" involves registering information about the URL or email address (or other address) through which each type of service is accessed. Finally, the question "How?" is answered by registering references to information about interfaces and other properties of a given service. These service properties describe how a particular software package or technical interface works.

UDDI API and Architecture. A UDDI programmer's API has been designed to provide a simple request/response mechanism that allows discovery of businesses, services, and technical service binding information. A set of Web Services supporting at least one part of the API is referred to as a UDDI node. One or more UDDI nodes may be combined to form a UDDI *registry*. The nodes in a UDDI registry collectively manage a particular set of UDDI data. The architecture of UDDI is hierarchical. It is up to each of the entities to register themselves upon one or several registries. There are mechanisms to replicate information concerning services or any other entity among nodes attached to a same registry. All the nodes of a same registry share the same information, thus when any modification occurs in one of these nodes, it has to be reflected on all the other nodes. Upon modification the node will issue a change notification to all the nodes attached to the same registry so that they reflect that change on their own copy of the shared data.

Until here, we have presented the context of the dissertation (service discovery in grid computing) and have illustrated it with a few examples of service discovery systems picked within traditional computational platforms. We now refine our study and state the problem we address.

1.3 Problem Statement

1.3.1 Infrastructure

Henceforth, we simply consider a set of geographically distributed computing devices, that we call *servers*. Each of these servers provides a particular set of binaries, libraries, or software components, brought together under the term *service*. As in any computational platform, the purpose is to make these services available to *clients*, which are desktop or laptop computers of users who need to do a computation ability missing on their own computer and/or requiring far more power than it is able to offer. Then, users have to specify their requirements following a certain formalism in a *query* request. As illustrated on Figure 1.1 and already described above, *Service Discovery* is a process

taking a query as a parameter and returning a set of available servers satisfying the requirements enounced in the query, along with the information allowing to use the service in a remote mode (basically the address of the server). What is behind the cloud of Figure 1.1 can be on a central, powerful computer, or a set of processors of a distributed platform.

1.3.2 User Requirements

Users can specify their needs, dealing with both software and hardware aspects. We now review the main aspects of a service for which a client can request some particular values.

- **The Service.** Users are basically looking for a service and specify it with a *name* under which this service is usually referred to as. For instance the `DGEMM` refers to a routine of the BLAS library [57]. Users may also want to discover any routine of the SUN S3L library [6], whose all names begin with "`S3L_`".
- **The Operating System.** Users often need a particular operating system, compatible with their data to be processed by the computation. Moreover, operating systems do not have the same characteristics and functionalities, inducing performance variations. They also do not offer the same security level. Examples are `Linux Fedora`, `MAC OS X`. A query from a user may be "any Linux", what can be specified with "`Linux*`".
- **The Processor.** Users may specify a particular processor on which they want the service to run, for instance to avoid to send miscoded data (endianess, 64/32 bits architectures) and loose precious time. Examples are `Power_PC`, `x86`.
- **Performance Requirements.** Users may require a minimum amount of memory, CPU or storage load under which they will not be satisfied. For instance, a user knows the memory amount needed by the service to complete the computation in a decent time. Examples of such requirements are `Memory > 2 Go` and `Free CPU > 50 %`.

To sum up, we want the service discovery system to support *multi-attribute* queries, *range* queries, and automatic completion of partial search strings.

1.3.3 Platform Characteristics

Recall that the service discovery service itself must be offered in the kind of platforms that are emerging today. We consider three types of nodes: **client** nodes, **server** nodes, and nodes taking part in the discovery process by maintaining the information on available services and answering to clients' requests. We call them service discovery agents, or simply **agent** nodes. We can consider two types of computing architectures:

- A platform in which clients, servers and agents are three distinct sets of nodes.
- A pure *peer-to-peer* computational grids, in which each node can be client, server, and also agent, at different times.

In both cases, agents are run on nodes of a platform having today's platform characteristics, *i.e.,* several properties that must be taken into account when designing a service discovery service. We now recall these properties:

1. **The Scale.** The number of servers and services, and of clients and requests, is very large. As a consequence, the load generated by the service discovery process is as large. In addition, as we already stated, no stable centralized powerful infrastructure is available. Thus this load must be distributed as much as possible, *i.e.,* among every node of the whole network. Algorithms underlying the discovery process have to be totally decentralized.

2. **The Heterogeneity.** Moreover, to avoid some nodes to be overloaded and avoid bottlenecks, one has to carefully and *fairly* distribute the load generated by the service discovery among all nodes involved in the process by taking into account their particular amount of power in terms of bandwidth, CPU and storage load.

3. **The Dynamic Nature.** In those platforms, nodes are constantly joining and leaving the network without notice. The service discovery must remain efficient as nodes performing it can leave at any time. In other words, the service discovery have to be *fault-tolerant*.

1.4 Solution Overview and Outline

This dissertation details the diverse aspects of a solution to the above-described problem. First note that traditional approaches that we described above can not be efficient in such platforms since they all require a stable set of powerful processors able to store the information and process requests. Because the *peer-to-peer* community provides purely-decentralized and fault-tolerant algorithms to retrieve information, peer-to-peer systems have been a new field of investigation for designers of grid middleware. Our solution also calls upon some *peer-to-peer* concepts. More precisely, our solution relies on the indexing of information on services available on the platform in a particular tree structure, namely a *trie*, also known as *Prefix Tree*, or *Lexicographic Tree*.

Distributed Trie-Based Structure. We have first developed a set of algorithms able to maintain such a structure and the information contained inside in a fully distributed environment, as services of the platform appear and disappear. These algorithms, their complexities, their performance measured by simulation and a first comparison of our solution with similar existing architectures are exposed in Chapter 3.

Mapping the Trie on the Network. The second main point focuses on how to improve the initial design presented in Chapter 3, in terms of throughput, *i.e.,* the amount of requests satisfied especially when this number becomes large. Chapter 4 presents a scheme to efficiently distribute the nodes and links of the tree structure on the processors, *i.e.,* an efficient *mapping* of the tree on the network. Chapter 4 also proposes some weighted load balancing techniques to dynamically adapt this mapping as the tree grows and the load, *i.e.,* the set of requests processed by a node, fluctuates. An analytical and simulation-based comparison with similar work is provided, establishing the relevance of our approach.

Fault-Tolerance and Self-Stabilization. The third main part of this dissertation is related to the fault-tolerance. The question we deal with in Chapter 5 is how one can ensure the effectiveness of the service discovery as nodes on which it runs are constantly joining and leaving the network. A first replication-based solution is proposed, replicating data, links, and nodes of the tree. Nevertheless, replication is costly and does not formally ensures the recovery of the system after arbitrary failures. From this point on, it remains only to use a *best-effort* approach, *i.e.,*

let the tree crash and recover a correct configuration with what remains in the network. To this end, we propose three algorithms, two of which rely on the self-stabilization paradigm. The concept of self-stabilization [54] is a general technique to design a system tolerating arbitrary transient faults. A self-stabilizing system, regardless of the initial states of the processors and initial messages in the links, is guaranteed to converge to the intended behavior in finite time. Our three algorithms are designed using different models and making different assumptions, and have different advantages and drawbacks in theory and practice. They are given with their complete correctness proofs and some simulation results to better capture their efficiency.

Middleware Extension, Prototype Implementation and Application. Finally, we detail our practical contributions in Chapter 6. First, we describe an early work connected to service discovery, whose aim was to identify and break the barriers to the scalability in a grid middleware. This work, partly independent from the architecture described throughout this dissertation was conducted in 2004, and was the design, implementation, and experimentation of a peer-to-peer extension of a particular grid middleware. Properly related to our architecture, we then present a prototype implementation of it. We focus on the software architecture we developed. Finally, we expose the use of such an architecture for the monitoring and provisioning of network resources in a distributed fashion and presents preliminary deployment results over the Grid'5000 platform.

The following chapter provides the background and related works for these problems, from the origins of the file retrieval techniques to the most advanced solutions merging self-stabilization and peer-to-peer systems via topology awareness and load balancing in distributed hash tables, and the tremendous amount of papers addressing the expressiveness of queries on top of structured peer-to-peer networks.

Chapter 2

Background and Related Work

In this chapter, we give the background and related work of our solution. In a first Section, we briefly expose the genesis of the convergence between grid computing and peer-to-peer systems and why peer-to-peer has become a field of investigation for designers of grid systems. Then, in Section 2.2, we recall the origins of the peer-to-peer systems and define some useful peer-to-peer concepts. In Section 2.3, we give some example of pioneering peer-to-peer systems, which rely on flooding algorithms. In Section 2.4, we detail the Distributed Hash Tables, their topology awareness mechanisms and the studies of their load balancing properties. In Section 2.5, we describe an important part of our related work, dealing with complex queries in structured peer-to-peer networks. In Section 2.6, we introduce a few recent advances in software considerations for the peer-to-peer and grid computing convergence. Finally, the background for self-stabilization and the recent studies for the use of such a paradigm in peer-to-peer networks is given in Section 2.7.

2.1 On the Convergence of Peer-to-Peer and Grid Computing

As new infrastructures hosting computational grids become larger and more unstable while failing to provide central devices able to monitor the whole platform, grid middleware needs to be redesigned in a fault-tolerant and fully decentralized fashion. Software introduced by the peer-to-peer community, offering purely decentralized techniques to share resources on highly large and dynamic platforms has appeared to be of high interest for grid middleware designers. Peer-to-peer technologies offer more and more robust tools for content distribution, while addressing failures. As a consequence, it has much to offer to the Grid Computing community. This convergence were put in words by Iamnitichi and Foster [83].

2.2 Peer-to-peer: Definitions and Origins

We can find in literature a considerable number of definitions of the term *peer-to-peer (P2P)*, each one encompassing a more or less broad set of systems.

- In their strictest definition, *pure* P2P systems refer to totally decentralized systems in which all basic entities, called *peers*, are equivalent and perform the same task. The KaZaA [146] network, within which *supernodes* (mini-servers locally aggregating information) are dynamically assigned, falls out of this definition.

- In a broader and widely accepted definition, notably introduced by Shirky [131], *P2P* refers to any *applications that take advantage of resources — storage, cycles, content, human presence — available at the edge of the Internet.* This definition encompasses systems completely relying on a central server for their operation, *e.g.,* volunteer computing systems (@home projects) — refer to Chapter 1, or pioneering file sharing networks like the well-known Napster [109].

File Sharing. The first generation of peer-to-peer file sharing networks relied on a centralized server listing the files available on users' computers. In this model, the user looking for a file sends a search request to this server and the server sends a response containing a list of online computers providing the requested file back to the user. The most known implementation of this model is Napster, a music file sharing system, which operated between June 1999 and July 2001. Napster encountered several problems.

- **Architectural Problems.** As the number of connected users increases, the central server becomes unable to manage the amount of requests, what reduces the ability of the system to scale well. The main weakness of the Napster system is that it relies on a central server. In case of a single failure on this server, the whole sharing system becomes inoperative.
- **Legal Problems.** The free sharing of music files led to accusations of massive copyright violation from the music industry. Although the original service was shut down by court order, it paved the way for decentralized peer-to-peer file-sharing programs, which have been much harder to control.

2.3 Unstructured P2P

The second generation of P2P file sharing systems, notably initiated among others by Gnutella [70], relies on a purely decentralized logical network built on top of the physical network in which all users participate in the forwarding of the search requests, each request flooding the network within a given radius (number of hops) from the source. However, Gnutella does not take advantage of the heterogeneous nature of file sharing networks in which some computers can be far more powerful than others in terms of computing power, storage space, or proximity to network backbones. KaZaA [146] improves this model by introducing a two-layer architecture. A set of *supernodes*, local servers dynamically assigned considering their power and listing the set of files stored by standard nodes, manages the listing and processes requests for other *standard* nodes, each *standard* nodes being assigned to one given supernode.

Despite the KaZaA effort to improve the architecture of unstructured approaches, such networks present several major drawbacks:

- **Overhead.** Flooding the network for each request generates a high traffic load that may be hard to handle and create poor network conditions. As the number of users grows, the number of messages can become unacceptable.
- **Non-Exhaustiveness.** The core operation of peer-to-peer systems is efficient location of data items. Scanning the entire network for each request would lead to unacceptable latencies. As a consequence, the search is limited by a given number of hops after which the packet is deleted. This means that there may be a response to the request somewhere in the network that has not been retrieved.

Addressing both high cost and non-exhaustiveness of unstructured peer-to-peer approaches for data item locations, a plethoric amount of work was initiated, by the famous Distributed Hash Tables.

2.4 Distributed Hash Tables

2.4.1 Principles

Distributed Hash Tables (DHTs) are very attractive solutions to distribute and retrieve information in a large and dynamic network. DHTs basically provide one single functionality to the client user, namely the location of data items.

In a hash table, each resource declared is referenced as a $(key, value)$ pair. For instance, one user may declare the resource (`musicFile1.ogg`, `140.55.123.4`) meaning that the computer at the address `140.55.123.4` provides the file `musicFile1.ogg`. These pairs are physically stored in m memory cells by application of the hashing function $h : X \to \{0, 1, \ldots, m-1\}$ on the key. In the case of distributed systems, the goal is to uniformly distribute the set of pairs among the m nodes of the network while globally optimizing the search for a key from any node in the network. As a result, the hash function must be uniform random with values in a set of logical identifiers with a negligible probability of collision. This is achieved by using cryptographic functions and large values for m.

Distributed Hash Tables maintain an *overlay network*, *i.e.,* they build a virtual communication network over a physical network, like the TCP/IP Internet protocol. Each processor of the physical network becomes a *node* of the overlay network. The location of the node representing a processor in the logical/overlay network is decided by applying the hash function on the unique physical address of the processor. A processor joins the overlay network by first using a *bootstrap* mechanism giving the way to access to a processor or a group of processors that are already in the overlay network. Once the joining processor knows an entry point of the system, it can send a request to be inserted. A new node is inserted using the distributed *join* procedure, finding its location in the overlay. Objects ($(key, value)$ pairs) are then distributed among these nodes by finding the node whose identifier (in the logical network) is the closest to the *key* of this object. The *lookup* function is a common feature of any DHT and allows to retrieve the node responsible for storing the searched key. We now briefly review the main pioneering implementations of such networks, developed around 2001.

2.4.2 Some DHTs

CAN. The overlay network maintained by the *Content-Addressable Network* [118] is arranged as a virtual d-dimensional Cartesian coordinate space on a d-torus. In other words, the hash functions returns values representing points in this space. When N nodes are in the network, the space is partitioned into N zones, each zone being assigned to a node. Figure 2.1 illustrates such an overlay for $N = 5$ nodes and $d = 2$. The first node to join owns the entire CAN space, its zone being the complete virtual space. When the second node joins, the space is split in two and each node gets one half of the space. Each node maintains a state of its neighbors *i.e.,* nodes responsible of zones adjacent to its own, allowing nodes to route messages to their destination node/zone. Consider a new node p joining the network. By applying the hash function on its IP address (for instance), it obtains a point h_p in the space and contacts any node already in the network. An insertion request for this node is then routed to the node responsible — let us denote it r — for the zone in which h_p falls. On receipt of this request, a node forward the request to the neighbor whose zone is the closest

to $h(p)$. Once found, the destination zone is split in two halves, one being assigned to p, the other to r. Objects are then assigned to nodes using the same routing algorithm.

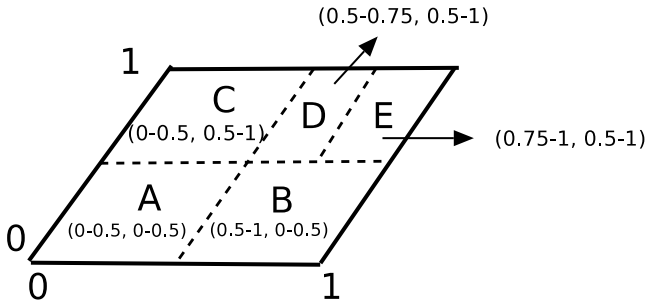


Figure 2.1: CAN, $d = 2$.

Chord. The Chord overlay network [135] builds a ring on identifiers ordered in an *identifier circle* modulo 2^m. A key k is assigned to the first node on the ring whose identifier is equal to or follows k in the identifier space. This node is called *successor* of k ($succ(k)$). In other words, each node is responsible for keys falling between itself and its predecessor on the ring. Such a logical ring is given by Figure 2.2 with $m = 3$ and the number of nodes $N = 4$ (black-filled circles). The routing table of a node identified by p is composed of references of nodes, called *fingers*, whose distance on the ring to the node grows exponentially, the ith entry being $succ(p + 2^i)$. To route a request for the key k, a node forwards the request to the highest finger lower than k.

Plaxton Trees, Tapestry and Pastry. The basic principle of routing methods used by Pastry and Tapestry DHTs relies on the Plaxton, Rajaraman, and Richa (PRR) routing scheme [114]. Each node receives a unique identifier of length d (let's say 160 bit) in a circle identifier. A request for a key k is routed on node p towards a node that shares at least one digit more with k than p does. As a consequence, each node is the root of a spanning tree (consisting of the paths leading down to this node). Pastry and Tapestry differ only in the method by which it handles network locality and replication. The Bamboo DHT [120] follows the same overlay construction pattern as Pastry and Tapestry but focuses on the churn problem. An open source DHT is based on this DHT [52].

Other DHTs. Many other DHTs have been introduced, based on other topologies and offering different improvements and flexibility levels. Among them, we can cite Koorde [89] and D2B [63] relying on De Bruijn graphs. Kademlia [105] proposes a routing scheme using the XOR metric between two identifiers. Viceroy creates a DHT using a butterfly topology overlay [101], and Cycloid [130] a cube-connected cycles overlay. Small Worlds [92] gives clues to build large overlay networks, based on the small world principle and Symphony [103] applies their results to the case of ring-overlay networks.

Complexities. Complexities of the previously mentioned DHTs are summarized in Table 2.1. The *Linkage* column gives the number of nodes each node maintains a connection with, *i.e.,* the size of the routing table, the *Update* column exposes the cost of an update of the network, *e.g.,* resulting from a

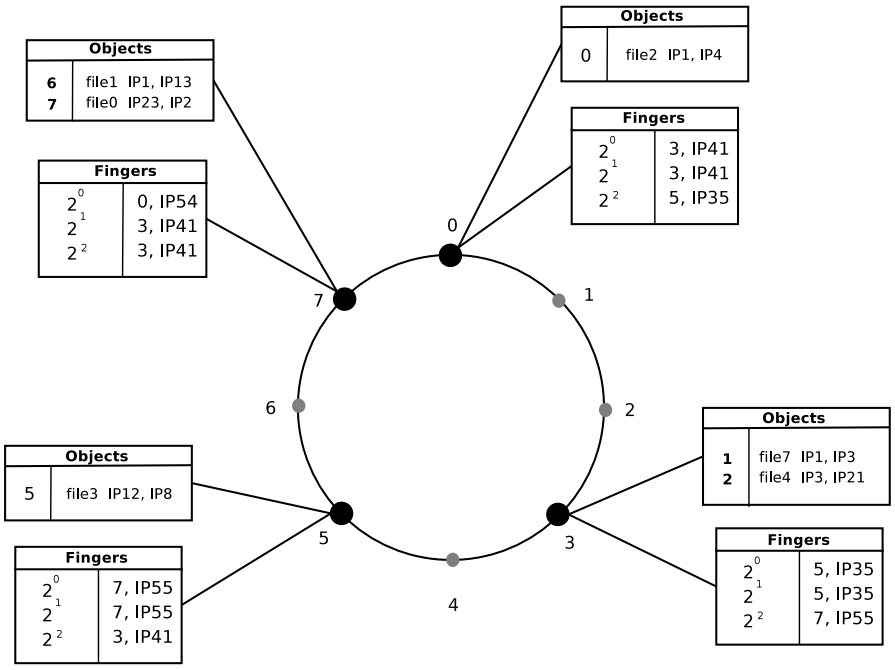


Figure 2.2: Chord, $m = 3$.

join operation, the *Lookup* column represents the number of hops required to find the node responsible for a key, and the *Congestion* column summarizes the expected load of one lookup operation on one node in a search for a random key starting from a random point. n denotes the number of nodes. d denotes both the dimension of the Cartesian space of CAN and the *network dimension* of Cycloid, where $n = d \times 2^d$. k is the number of long links in Symphony.

Other Properties. By design, DHTs have several good properties.

- **Load Balancing.** Uniform random hash ensures with high probability the uniform distribution of keys among nodes, each node being responsible of the same amount of keys.
- **Fault Tolerance.** Several mechanisms ensure the consistency of the overlay and prevent from data loss as nodes are leaving the network without notice. Periodic scanning allows to detect that a node has left. By introducing a replication mechanism — each object being replicated k times, the system ensures that $k-1$ simultaneous will not lead to the loss of any data. Tuning k should be the best tradeoff between storage space requirements and fault tolerance.

In the remainder of this section, we focus on two problems which are inherent to the DHT design, but that may greatly limit their use on real platforms, namely *topology awareness* and *load balancing*.

2.4.3 Topology Awareness

As DHT overlays are built on random logical connections, they break the *natural* topology of the physical network. It is important to recall that neighbors in the overlay network may be a very long way from one another. As a consequence, it is possible to achieve a world tour even if the destination

	Linkage	Update	Lookup	Congestion
CAN [118]	$O(d)$	$O(d)$	$O(dn^{1/d})$	$O(dn^{1/d-1})$
Chord [135]	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O((\log n)/n)$
Pastry [121]	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O((\log n)/n)$
Tapestry [159]	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O((\log n)/n)$
D2B [63]	$O(1)$	$O(1)$	$O(\log n)$	$O((\log n)/n)$
Koorde [89]	$O(1)$	$O(1)$	$O(\log n)$	$O((\log n)/n)$
Kademlia [105]	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O((\log n)/n)$
Small Worlds [92]	$O(1)$	$O(1)$	$O(\log^2 n)$	$O((\log^2 n)/n)$
Symphony [103]	$O(k)$	$O(k)$	$O((log^2 n)/k)$	$O((log^2 n)/(nk))$
Viceroy [101]	7	$O(1)$	$O(\log n)$	$O((\log n)/n)$
Cycloid [130]	7	$O(1)$	$O(d)$	$O(d/n)$

Table 2.1: *Expected* performance measures of DHTs.

node is in the same cluster as the initial node. As we will see now, this aspect of DHTs caused a lot of work to be done to inject *topology awareness* in the way overlays are built.

In order to improve the performance of DHT overlays, many works proposed approaches to make the logical connectivity congruent with the underlying (IP-level) topology. In other words, their goal is to build a logical network in which neighbors are actually closed in terms of latency, administrative domains, or according to any *proximity* metric, in the IP network. Most of approaches studied rely on a hierarchical design, but we can classify the set of approaches in two distinct groups: *supernodes*-based and *landmarks*-based.

Supernodes-Based Approaches. Brocade [158] is a two-layer DHT. Its key idea is to let each administrative domain build its own DHT, in which one or several leaders are elected considering characteristics like CPU power, bandwidth, proximity with a backbone or fanout. These leaders then enter an *interdomain* DHT, connecting local DHTs by high-speed links. This concept has been generalized to any levels [66], and adapted to the IP numbering [68]. The work presented in [152] extends the flexibility of the supernodes approach. When a node joins the network, it chooses itself to be a supernode or not and selects a few existing supernodes to be its neighbors according to some globally known policies. Jelly [81] proposes an architecture in which each joining node either meets the requirements to be a supernode or chooses to be a child of the supernode that minimizes a given metric.

Landmarks-Based Approaches. Another kind of approach, introduced in CAN [118] consists in using a set of stable nodes distributed on the physical network, called *landmarks*. These landmarks are used to dispatch nodes in virtual *bins* considering a given metric, such as the latency. The algorithm used to make the distribution is as follows: Let us consider m landmarks. $m!$ orderings on the set of landmarks are possible, each ordering corresponding to one bin. Each node computes the latency between itself and each landmark, sorts them and thus finds its own bin. The intuition behind doing this is that nodes that are close to each other have similar landmark measurements. ECAN [155] extends this approach by building a hierarchy of bins, grouping two close bins at level 1 in the same bin of level 2, and so on. Finally, some papers, like HIERAS [153] adopt a hybrid approach between landmark and supernodes. The Expressway network [152] elects supernodes to

which ordinary nodes are connected. Each supernode also measures its *network distance* to a set of m landmarks, what gives a vector of distances and a unique position in a m-dimension Cartesian space. This space is finally reduced to a 1-dimension space using a preserving-locality scheme relying on space filling curves.

Hierarchical DHTs call upon network management tools (autonomous systems, local administrative domains, IP numbering) and strongly rely on the assumptions that local connections are always quicker than long distance connections. Landmarks-based approaches assume a globally known set of stable nodes that are distributed in a way making measurements relevant. To conclude, even greatly improving the performance of the overlay network, these approaches still assume a minimum stability of infrastructures.

2.4.4 Load Balancing

In pioneering DHTs we have described before, the attempt to balance the load fails in several ways. First, the random assignment of items leads to an $O(\log n)$ unbalance factor in the number of object stored at a node. In other words, some nodes receive more than their fair share, by as much as a factor of $O(\log n)$ times the average. Second, many applications bypass the uniform assignment, for instance to easily support range queries, using order-preserving hash functions. We now review the tremendous amount of the work this issue produced.

Virtual Servers. First introduced by Chord [135] to reduce the load unbalance, the notion of virtual server represents a node in the DHT, responsible for a contiguous region of the DHT's identifier space. Then, each processor can own multiple ($O(\log n)$) non-contiguous regions of the space by hosting multiple virtual servers. This allows a reduction of the unbalance factor from $O(\log n)$ to $O(\log n / \log\log n)$.

The Power of Two Choices. A lot of solutions to the load balancing problem relies on the *Power of Two Choices* paradigm. The strength of this paradigm was established by Azar, Broder, Karlin, and Upfal [24]. Let us consider the basic balls and bins problem, each of n balls choosing one of n bins independently and uniformly at random. Then the *maximum load*, *i.e.,* the largest number of balls in any bin is approximately $\log n / \log\log n$. Now suppose instead that the balls are placed sequentially, each ball being placed in the least loaded of $d \geq 2$ randomly chosen bins. Then the maximum load is $\log\log n / \log d + \Theta(1)$. This result implies two important things. The first implication is the fact that $d = 2$ leads to an important reduction of the maximum load. The second one is that any choice for d beyond 2 decreases this maximum load only by a constant factor. The application of this result to the load balancing problem in DHT network is almost straightforward. Bins are nodes, balls are items, d is the number of possible locations for a joining node. Byers, Considine and Mitzenmacher [35] have built the first load balancing algorithm relying on the power of two choices paradigm for DHT networks. Basically, in this algorithm, when a node joins, it chooses the best location (the one locally providing the best load balance) among a set of locations given by different hash functions.

Address-Space Balancing and Item Balancing. Independently, Karger and Ruhl proposed an algorithm that improves the load balance of Chord-like networks by giving the possibility for each node to choose dynamically (each time it is required) the best position among $\log n$ possible ones.

The authors prove that using such a scheme requires $O(\log \log n)$ address changes upon insertion or deletion to reach an optimal state, *i.e.,* each node stores an $O(1/n)$ fraction of the entire key-space with high probability. However, Address-Space balancing is not enough if the initial hash function is not random but for instance preserves the order of keys. In this case, the authors propose item balancing to be explicitly done, when nodes join and leave the network. Consider a node i with a load ℓ_i. Periodically, it contacts a random node j with load ℓ_j. Load balancing process is launched if the ratio between ℓ_i and ℓ_j is lower than a global constant ϵ / $0 \leq \epsilon < 1$. Assume $\ell_i > \ell_j$. We distinguish two cases:

1. $i = j+1$: (i and j are neighbors) then j increases its address until each node is loaded $(\ell_i+\ell_j)/2$.
2. $i \neq j+1$: If $\ell_{j+1} > \ell_i$, i becomes the new successor of j and the same process applies. Otherwise, j moves between i and $i-1$ to capture half of i's items.

The authors prove that by repeatedly applying this process, for any node i, ℓ_i is always within a constant factor of the optimal. Naor and Wieder [108] introduced a similar technique in their *Continuous-Discrete approach*, but their algorithm works only in their particular overlay.

Continuing the Work. Bienkowski *et al.* [31] propose a similar scheme applying to any overlay — length of intervals assigned to nodes differ at most by a constant factor. Their scheme achieves optimal load balancing in a constant number of *rounds*, each round having a complexity in $\Theta(\mathcal{D} + \log n)$ where $\mathcal{D}$ is the diameter of a specific network (most of the time, $\mathcal{D} = \Theta(\log n)$ or $\mathcal{D} = \Theta(\log n / \log \log n)$). While Karger and Ruhl [90] requires $O(\log \log n)$ re-assignments of nodes for each arrival/departure, their scheme achieves a number of re-assignments within a constant factor from optimal centralized algorithm. Other algorithms established still more precise bounds for the ratio between the largest and the smallest amount of items each node manages, denoted σ. Manku [102] obtains $\sigma \leq 4$ requiring a message cost of $\Theta(\mathcal{D} \log(n))$. Giakkoupis and Hadzilacos [69] also obtains $\sigma = 4$, requiring a cost in $\Theta(\mathcal{D} + \log n)$. Finally, Kenthapadi and Manku's goal [91] is to find the best tradeoff between local and random probes. Denoting v the number of local probes and r the number of random probes, they establish that for any combination of r and v such that $rv \geq c \log n$, $\sigma \leq 8$. These results, partly summarized in [91] are presented in Table 2.2.

The Heterogeneity Problem. The uniform distribution of keys among nodes leads to a *good* load balancing *only if* we assume the homogeneity of at least two parameters, namely (i) the **capacity of nodes**, *i.e.,* the number of requests each node is able to process, and (ii) the **popularity of items**, *i.e.,* the number of requests on each item the system receives. The paper of Saroiu *et al.* [123] presents measurement studies highlighting this heterogeneity, in terms of bandwidth, storage and CPU. The assignment of an even number of items to every node makes sense only if the capacity of nodes are the same. If we assign more items to a node than it can handle, and/or give this same amount to another one whose capacity is higher, the throughput is clearly lower than it could be. Moreover, the assignment of the same number of keys to each node results in a uniform load balancing **only** if the keys are also uniformly requested. This can not be ensured in any real system, which depends on users' requests. Godfrey *et al.* [71] present a load balancing mechanism in which heterogeneity of nodes is taken into account by using virtual servers. At each load balancing round, some temporary *masters* are elected to gather the load information and compute the best redistribution according to a metric. The drawback of this approach is clearly its semi-centralized fashion, in the sense that master nodes must remain active during the computation. Zhu and Hu [160] proposes an attempt

	σ	Message Cost	comments
Naor & Wieder [108]	$\Theta(1)$	$\Theta(\mathcal{D}\log n)$	Works only for the Continuous-Discrete Overlay
Karger & Ruhl [90]	$\Theta(1)$	$\Theta(\mathcal{D}\log n)$	$\Theta(\log\log n)$ reassignments are required
Bienkowski *et al.* [31]	$\Theta(1)$	$\Theta(\mathcal{D}+\log n)$	Cst factor to the *optimal* number of reassignments
Giakkoupis & Hadzilacos [69]	4	$\Theta(\mathcal{D}\log n)$	1 reassignment is required
Manku [102]	4	$\Theta(\mathcal{D}+\log n)$	1 reassignment is required
Kenthapadi & Manku [91]	8	$\Theta(r\mathcal{D}+v)$	Tradeoff between r and v, $rv \geq c\log n$

Table 2.2: Algorithms for improving the load balancing in dynamic DHTs: σ is the unbalance factor between the most and the least loaded nodes, the second column gives the total message complexity of the schemes, and the third column recalls some important features. $\mathcal{D}$ denotes the diameter of the underlying overlay network, v and r the sizes of local and random probes, respectively, and c is a small constant.

at solving the issue of load balancing among heterogeneous DHTs while injecting some proximity awareness to reduce the cost of reassignments of objects. To this end, they combine several existing mechanisms. First, they use virtual servers that are dispatched among physical nodes. A k-ary tree is built to gather load information and make decision for the next reassignment. Finally, the reassignment is done by heavy loaded nodes giving some work to *close* light nodes. This *closeness* is computed by using the *landmark* approach. Godfrey and Stoica [72] extends the Chord protocol by adapting the virtual servers paradigm to the heterogeneous case and prove that their scheme achieves a load unbalance factor arbitrarily small. Ledlie and Seltzer [95] extends the random choice approaches to take into account heterogeneity of both nodes capacity and object load. In 2008, Fu *et al.* [64] generalizes the initial power of two choices results. In particular, they establish that the maximum load in a P2P environments with heterogeneous nodes capacity and node churn for $d \geq 2$ random choices is $c\log n/\log d + O(1)$ where c is a constant.

DHTs allow to retrieve resources by only one attribute, used to generate the key. In other words, DHTs only support exact-match queries, drastically reducing the expressiveness of requests and thus offering very limited facilities for the user to specify (and retrieve) what he is looking for. For instance, a user may want to discover any files whose filename begins with a given string or make multi-attribute queries. As detailed in the next section, a lot of work has intended to make DHTs querying possibilities more flexible and complex.

2.5 Complex Queries

2.5.1 First Improvements

Harren *et al.* [75] give first clues to support complex queries system on top of structured P2P systems in terms of architecture, mainly by proposing to add a *Query Processing* layer aiming at supporting more complex queries on top of the basic lookup. To build this upper layer, a series of work proposes simple extensions of existing DHT lookup techniques. INS-Twine [100] allows the retrieval of objects described by semi-structured descriptions, for instance using XML. Biersack *et al.* [67] describes a

similar scheme but distributing the resolution of the request on several nodes for a better scalability. In the paper by Triantafillou and Pitoura [113], this upper layer allows to plug traditional database operations (select, join, etc.) on top of DHTs. One problem which generated a high number of papers is the enhancement of DHTs with *Multi-Attribute Range Queries*. We now focus on this issue.

2.5.2 Range Queries

A user interested in certain resources issues a query that is a combination of desired attributes values or their desired range. A typical example can be:

```
Binary == DGEMM && OS == Linux && Memory >= 512 && Storage >= 4096
```

In this case, the system will search for a DGEMM operation (matrix multiplication) to be run on Linux and providing at least 512 MByte of memory and 4 GByte of storage space.

Composing Single Lookups. The first attempt at enabling range queries was based on simple extensions of existing lookup mechanisms. Among them, MAAN [36] (*Multi-Attribute Addressable Network*) extends Chord using a uniform locality-preserving hashing to map attribute values to the Chord identifier space. Mercury [30] directly operates on an attribute space, with random sampling to provide a more efficient query routing and load balancing. Though both MAAN and Mercury provides range queries, they achieve it mainly through composing single-attribute query resolution. XenoSearch [134] builds a similar system on top of the Pastry overlay. Finally, Sword [112], based on the same principle, builds an information service able to locate a set of machines matching user-specified constraints on both static and dynamic node characteristics, including both single-node and inter-node characteristics.

Inverse Space Filling Curves - Andrzejak and Xu. Andrzejak and Xu [17] use the **inverse** Hilbert mapping to map a single attribute domain onto CAN's d-dimensional Cartesian space. A Space Filling Curve (SFC) [20] is a continuous mapping from a d-dimensional space to a 1-dimensional space. Imagine a d-dimensional cube with the SFC passing through each point in the cube volume, and entering and exiting the cube only once. Given a point of a d-dimensional space, the SFC returns a real value between 0 and 1, while preserving the locality, *i.e.,* close points in the d-dimensional space will have close values when projected on the unit vector. **Inverse** SFCs associate any value in the interval [0,1] to a cell of a d-dimensional space. As illustrated on Figure 2.3 with the Hilbert SFC, they can be refined recursively — here, the refinement level is 2. In the system proposed [17], the refinement level is globally known allowing any node to locally find which cell is responsible for a given value or a range of values. The last step of the process is illustrated by Figure 2.4. The d-dimensional space is mapped on the d-torus maintained by CAN, each node storing the values that falls in cells included in their zone.

Trie-structured Overlays. An alternative set of approaches for efficient support of range queries were introduced, relying on another kind of overlay, namely, *tries*. A *trie* is a particular kind of rooted tree for storing strings in which there is one node for every common prefix. For instance, each node of a *binary* trie is labeled by a prefix that is defined recursively: given a node with label l, its left and right child nodes are labeled $l0$ and $l1$, respectively.

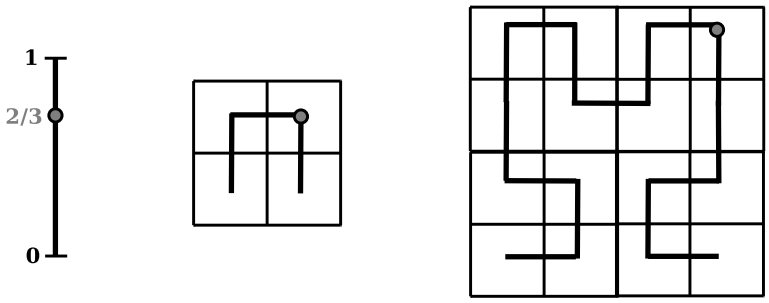


Figure 2.3: Inverse Hilbert Space Filling Curve refinements. The 2/3 value is in cell $(.5, .75)$ at first refinement and in cell $(\frac{10}{16}, \frac{11}{16})$ at second refinement.

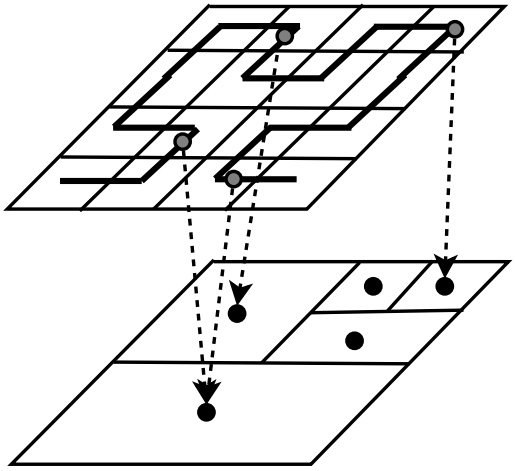


Figure 2.4: Mapping the values (grey-filled points) on the nodes (black points) of the CAN network.

Skip Graphs. Skip Graphs [21] generalizes skip lists for distributed environments and is a sort of *probabilistic* trie. Each node in skip graphs is a member of multiple doubly-liked lists at several levels. As illustrated on Figure 2.5, the bottom-level consists of all nodes in increasing order of keys. A membership vector, randomly generated for each node, of size in $O(\log n)$, determines the lists to which a node belongs. For instance, a node is in the list L_w at level i if and only if w is the prefix of length i of its membership vector. The process of seeking a key starts at the top-most level. The request is processed along the same level without overshooting the key. If no node was found on this level, the process continues at a lower level, possibly reaching level 0. Each node maintains an average of $\Theta(\log n)$ neighbors and skip graphs support $O(\log n)$ search time. Range queries are supported in $O(\log m)$ time and $O(m \log n)$ messages, where m is the number of nodes pertained by the range. The authors provide an analysis of the load balancing properties of such a structure. If one given key is constantly requested, the overload on this node can not be avoided. However, this effect drops off rapidly with distance to this node. More precisely, the probability of a node u to be passed through by a request for k is inversely proportional to the distance between u and k. Brushwood [157] focuses on preserving data locality and proposes a scheme linearizing tree-structured data and mapping it on a Skip

Graph network.

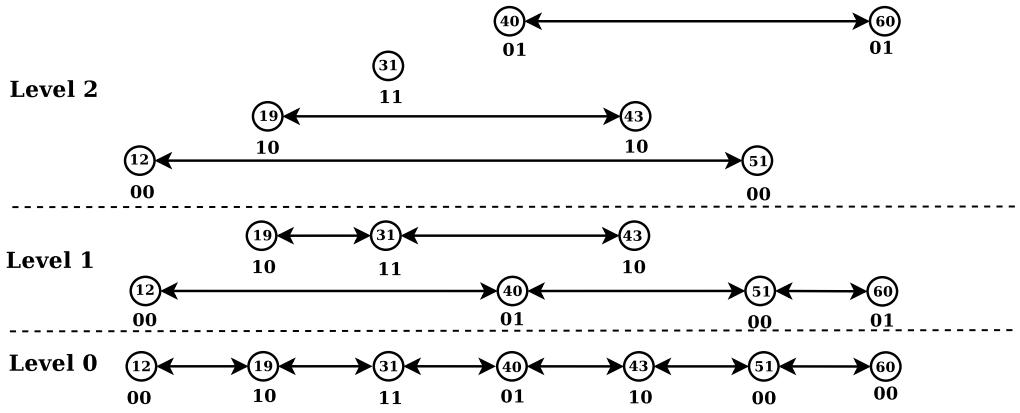


Figure 2.5: A skip graph with 3 levels.

Prefix Hash Tree. Prefix Hash Tree (PHT) [117] basically builds a binary trie over the data set. Figure 2.6 gives a sample of a PHT network. Each leaf node of the trie stores the keys prefixed by their own identifier and maintains a pointer to the leaf node on its immediate left and immediate right, respectively. PHT is a two-layer architecture in which each trie node (internal nodes and leaf nodes) is mapped on the network through a DHT. A leaf node with label l is thus assigned to the DHT node to which l is mapped by the DHT, *i.e.,* the DHT node whose identifier is the closest to $h(l)$. The PHT lookup operation on one key k returns the leaf node storing it, denoted $leaf(k)$. The PHT lookup is basically a set of DHT lookups, launched using a dichotomic search on the size of the key. For instance, to locate the leaf node responsible for the key 001100, a first DHT lookup is launched on the key 001. As illustrated on the trie of Figure 2.6, this trie node is not a leaf node, so another DHT lookup is performed on the string 00110, which is still not a node. Finally, the DHT lookup on node 0011 finds the corresponding leaf node, which sends responses back. A PHT lookup clearly requires $\log D$ DHT lookups, where D is the maximum size of the keys, leading to a total complexity in $O(\log D. \log N)$, where N is the number of nodes of the DHT. Range queries between two values L and H require two steps. First, a PHT lookup is performed to find the trie node whose label is the greatest common prefix of L and H (or the smallest prefix range that covers the query). On Figure 2.6, the sought trie node for the range $[000110; 001100]$ is $00*$. Then, the request is propagated in the subtree rooted at $00*$ in parallel, leading to a latency in $O(D)$. The load balancing relies on a global threshold above which leaf nodes are split into two child nodes, keys being stored on the child with the corresponding prefix.

P-Grid P-Grid [48] is a similar trie-structured overlay in which each *peer* (*i.e.,* a processor of the physical network) $p \in P$ is associated with a leaf node of a binary trie. Each leaf corresponds to a string $\pi \in \Pi$, the entire key space partition. An example of a P-Grid is given on Figure 2.7. Each P-Grid peer p, labeled $\pi(p)$ maintains a set of pointers (solid lines on Figure 2.7) to peers sharing a common prefix of different sizes with $\pi(p)$. More formally, for each prefix $\pi(p, l)$ of $\pi(p)$ of length l with $0 < l < |\pi(p)|$, p maintains a pointer to a peer satisfying the property $\pi(p, l) = \overline{\pi(p, l)}$, where $\overline{\pi}$ is π with the last bit inverted. Thus the cost for storing

these pointers are bounded by the depth of the trie. The P-Grid structure is very similar to the Kademlia overlay [105]. To route a request on k, a node finds among its references the node that prefixes k at most. For fault-tolerance multiple peers may be associated with the same key space partition. Consistency between replicas is maintained by epidemic protocols. The authors of P-Grid have developed a set of algorithm to periodically check the load balance in the tree, keys being split or exchanged to satisfy some global load balance properties [8]. A software based on this overlay is available on the web [116].

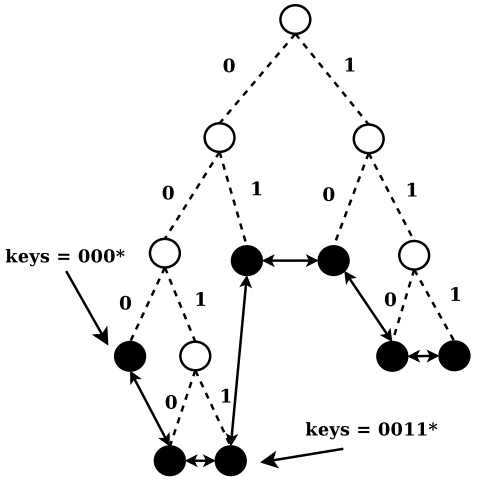


Figure 2.6: Prefix Hash Tree.

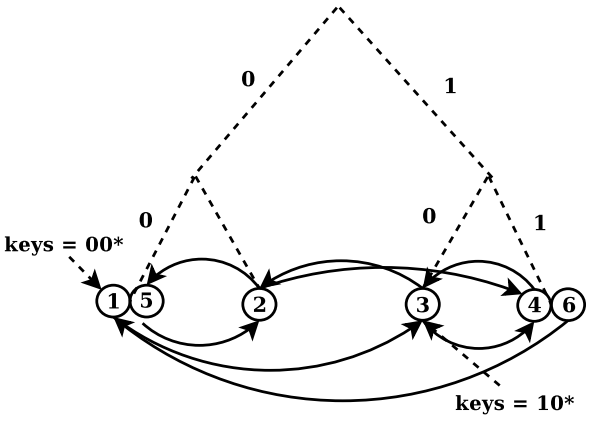


Figure 2.7: P-Grid.

All these approaches are able to achieve multi-attribute queries by simple extensions. These extensions mainly consist in maintaining one overlay per type of attribute, or *dimension*, and launching one query for each attribute. Basically, complexities are multiplied by the number of dimensions of the system. When this number grows, the scalability issue becomes more significant. Dealing with this point, other approaches propose *native* multi-dimensional solutions.

2.5.3 Multi-Dimensional Overlay

Space Filling Curves. Squid [125] *natively* supports multi-dimensional range queries and automatic completion of partial search strings. Consider each object, or *service* to be described by a fixed number of keywords. As a consequence, in the multi-dimensional keyword space, each dimension following the alphabetic order, each service is a precise point of the space, as illustrated by Figure 2.8. By applying the SFC mapping to a point, they obtain a unique identifier, ready to be placed on the underlying overlay — Chord in this particular paper. Queries consist of a combinations of keywords, partial keywords and wildcards. A query corresponds to some *clusters* of the data space. The cluster in light grey on Figure 2.8 identifies the zone of the data space pertained by the query $(00*, 01*)$. By applying the SFC mapping on the query, we find the identifiers corresponding to the request. (Here, identifiers that shares the common prefix $0011*$.) The query is finally sent to the nodes of the Chord ring responsible for these identifiers.

Similar approaches and some studies of their potential improvements have been proposed in [65, 97, 132]. The SCRAP system [132] is very similar to Squid. As both SCRAP and Squid partition

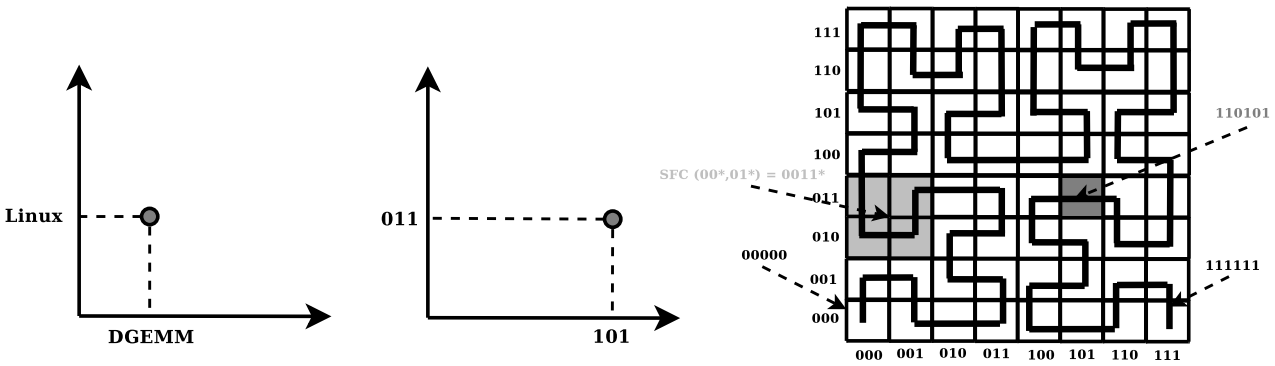


Figure 2.8: Squid. On the left, the service is described by `DGEMM` and `Linux`. In the middle, the object is described by keywords `101` and `011`. On the right, the Hilbert SFC mapping associates the coordinates $(101, 011)$ to the value 110101 and the query $(00*, 01*)$ to the cluster $0011*$.

the space statically (the partitioning level needs to be decided beforehand), ZNet [65] partitions the space dynamically and focuses on improving query efficiency and load balancing of such approaches and relies on a Skip Graphs overlay.

Nodewiz. Nodewiz [26] dynamically splits a multi-dimensional data space using a tree abstraction, each subtree corresponding to a subspace of the multi-dimensional space, for instance (`Mem > 2 && Load >= 0,6`). The main drawback of Nodewiz is that it assumes a set of reliable static nodes to host the system.

2.6 Computational Grids and Peer-to-Peer Concepts: Software Considerations.

Several projects recently propose prototype of grid middleware using some peer-to-peer concepts.

Vigne [88] is a prototype, whose goal is to ease the use of computing resources in a grid for executing distributed applications. Vigne is made up of a set of operating system services based on a peer-to-peer infrastructure. This infrastructure currently implements a structured overlay network inspired from the Pastry DHT. On top of the structured overlay network, a transparent data sharing service based on the sequential consistency model has been implemented. It is able to handle an arbitrary number of simultaneous reconfigurations. An application execution management service has also been implemented including resource discovery, resource allocation and application monitoring services.

Vishwa [119] provides a framework to execute distributed applications with synchronization dependencies by using the concept of distributed pipes that requires the modification of the applications.

Zorilla [161] is prototype Peer-to-Peer(P2P) grid middleware system. It strives to implement all functionality needed to run applications on a grid in a fully distributed manner, such as scheduling, file transfer and security. It provides locality-aware co-allocation with a mechanism called flooding scheduling. The resource discovery relies on gossiping [59].

The Arigatoni overlay network [96] aims at grouping a large set of individual computation abilities in a global hierarchical unstructured overlay using the concept of *colonies* (individuals ruled by some

imposed or elected leader) in a self-organizing manner. The project also focuses on resource discovery inside such an overlay [44] using an unstructured fashion and on the bases for a *programmable* overlay.

2.7 Fault-tolerance and Self-Stabilization

2.7.1 P2P Traditional Approaches Limits

Although fault-tolerance is a mandatory feature of systems targeted for large scale platforms (to avoid data loss and to ensure proper routing), trie-based overlays offer only a poor robustness in dynamic environment. The crash of one or several nodes may lead to the loss of information stored, and may split the trie into several subtries. These subtries may not be re-grouped correctly, making the system unable to correctly process queries. In recent trie-based approaches, the fault-tolerance is either ignored, or handled by preventive mechanisms, usually by replication, which can be very costly in terms of computing and storage resources. Afterward, the purpose is to compute the right trade-off between the replication cost and the robustness of the system. However, replication does not ensure the recovery of the system from arbitrary failures and may propagate a wrongly initialized variable, as it does not check the correctness of the variables replicated.

2.7.2 Self-Stabilization

Self-stabilization [53, 54] is a general technique to design a system tolerating arbitrary transient faults. A self-stabilizing system, regardless of the initial states of the processors and initial messages in the links, is guaranteed to converge to the intended behavior in finite time. Thus, a self-stabilizing system does not need to be reinitialized and is able to recover from transient failures by itself. In other words, imagine a program in which variables are set randomly. Self-stabilization claims that there is no need for a proper reset of the variables, as a self-stabilizing algorithm is guaranteed to recover *automatically* from an arbitrary initialization in a finite time. It appears an efficient technique suitable for dynamic, failure prone systems like peer-to-peer systems. We now present the background of any self-stabilizing work, assumptions traditionally made, different scheduling and communication models. We illustrate them by explaining some self-stabilizing algorithms. We end this chapter by describing some self-stabilization works especially conducted to address peer-to-peer systems.

2.7.3 System Assumptions

Distributed System. Consider a graph $G = (V, E)$ where V is a set of nodes (Vertices) and E is a set of links (Edges) between nodes. A network is said *static* if its communication topology remains fixed. A network is said *dynamic* if links and nodes can go down and recover later. Traditionally, it is assumed that the topology always remains at least weakly-connected, since otherwise, we have several independent networks. Algorithms are modeled as state machines performing a sequence of steps. One step consists of reading input and local state, and, depending on both the input and local state, performing some action. The state of a node is defined by the values of its variables. Performing an action on one node or not depends on the state of a node and its input at the beginning of the step. If the state of the node and its input triggers some action according to the protocol, the node is said to be *enabled*. The action results in a state transition and writing output. Communication between nodes can be done by several means, for instance by exchanging messages. In Dijkstra's theoretical model [53], in each computation step, each node can atomically read variables (or registers) owned

by its neighboring nodes. More precisely, one commonly used model for communications is that of shared memory [54], in which each pair of neighbors communicate by writing and reading dedicated registers. The grain of *atomicity* may vary [55]. *Composite atomicity* allows one node to read all input variables, make the state transition and write all its output variables in one atomic operation. *Read/write atomicity* only allows to **either** read **or** write its communication variables in one single atomic operation. Depending on the type of *scheduler*, or *daemon*, one step of the system may allow only one node to perform a state transition (central daemon), or several enabled nodes concurrently (distributed daemon). Finally, the scheduler can be of different levels of *fairness*. For instance, if the scheduler is unfair, even if a node p is continuously enabled, then p may never execute its pending action until p is the only remaining enabled node in the system.

Proving Self-Stabilization. The product of the states of all nodes is called the *configuration* of the system. Among the set of possible configurations, some are defined as *correct*. When proving an algorithm to be self-stabilizing, two parts are required.

1. **Convergence.** Starting from any configuration, by executing the algorithm, the system reaches a correct configuration.
2. **Closure.** Starting from any correct configuration, by executing the algorithm, the system remains in a correct configuration.

One point to have in mind is the fact that convergence refers to the time after the *final* failure occurred, since no convergence can be proved if some failures constantly appear. In other words, stabilization can only be guaranteed *eventually*, after the delay between two faults have become *long enough* to allow convergence.

2.7.4 Self-Stabilizing Trees.

In the self-stabilizing area, many investigations take interest in maintaining distributed data structures, in particular trees. Solutions in [77, 78, 79] focus on binary heaps and 2-3 trees. Several approaches have also been considered for a distributed spanning tree maintenance, *e.g.,* [18, 22, 46, 55].

A spanning tree of $G = (V, E)$ is a new graph $T = (V, E')$ consisting of the set of nodes of G connected by a set of edges $E' \subseteq E$ such that there exists exactly one path between each pair of nodes. Note that $|E'| = n - 1$. In a network in which broadcast operations are frequent, finding a spanning tree can significantly reduce the cost of using the tree, especially if $|E|$ is much larger than $n - 1$.

The algorithm presented by Dolev, Israeli, and Moran [55] is written in the *shared memory* model, *i.e.,* each node P_i has a set of ordered neighbors. Read and write operations are assumed atomic. P_i communicates with its neighbor P_j using two shared registers r_{ij}, in which P_i writes and from which P_j reads, and r_{ji}, in which P_j writes and from which P_i reads. Among the set of nodes, one node is assumed to be *special* and is the root of the tree. Registers are made of two parts: one specifying the distance of the writer to the root, and one containing a boolean specifying if the writer is the parent of the reader in the spanning tree or not (1 if it is the case, 0 otherwise). The algorithm is similar to a BFS (Breadth-First Search) and works as follows: the root writes 0 in the *distance* part of each of its registers in which it writes. Each other node p periodically reads the registers in which its neighbors write their distance to the root, and selects the neighbor with the minimum distance d. If several neighbors have the same distance d to the root, it keeps the one (let's say q) with the

smallest ordering value and writes 1 in the *parent* part of the r_{pq} register and $d+1$ in the *distance* part. In other registers in which p writes, p sets the *distance* part to $d+1$ and the *parent* part to 0. The spanning tree can be drawn reading the *parent* part of all registers.

A similar algorithm was published by Afek, Kutten and Yung [9], except that instead of assuming a predetermined *special* node to be the root, all nodes are uniquely identified and can be totally ordered. The root is the node with the largest identifier. They also use the same read/write atomicity model. Arora and Gouda [18] also propose a similar BFS-based self-stabilizing spanning tree, but using the composite atomicity model. Chen, Yu and Huang [46] also propose a self-stabilizing spanning tree protocol but whose result is not necessarily a BFS tree, because they use the scheduler to choose a new parent after breaking cycles, and so introduce a non-deterministic behavior.

2.7.5 Snap-Stabilization

The concept of *Snap-stabilization* was introduced in [34, 47]. A *snap-stabilizing* algorithm guarantees that it always behaves according to its specification. In other words, a snap-stabilizing algorithm is also a self-stabilizing algorithm which stabilizes in 0 steps. Note that the stabilization time is no more related to the configuration of the system but to the execution of the algorithm. In [34], authors propose a snap-stabilizing version of the *Propagation of Information with Feedback* (PIF) which is a wave algorithm gathering information in tree networks. In [29], the authors present the first snap-stabilizing distributed solution for the Binary Search Tree (BST) problem. Their solution requires $O(n)$ rounds to build the BST, which is proved to be asymptotically optimal for this problem in the same paper.

Snap-stabilization has first been introduced for the traditional coarse-grained communication model of Dijkstra [53]. The straightforward division of each high-atomicity action into a sequence of low-atomicity actions (like message exchanges) does not suffice to make the protocol working in a model with low-atomicity [110]. Until recently, it was impossible to prove snap-stabilization in a message passing environment. Recent works have made further investigations for the snap-stabilization in message passing [50], making it possible under specific constraints given by the authors.

2.7.6 Self-Stabilization for Peer-to-Peer Networks

Limits of Traditional Self-Stabilization Models. All these solutions are designed for distributed systems defined by their topology, each node having a set of neighbors, communicating with them through a finite number of links. In today's emerging platforms, like the Internet, each processor $P1$ can communicate with any other processor $P2$, provided that $P1$ knows the address of $P2$. Similarly, in front of my computer, I can consult any webpage, provided I know its address. The topology of P2P networks, or more globally of high-level protocols are logical, built on top of the physical network. Details of the physical topology, and the underlying routing process are abstracted. In other words, in a peer-to-peer overlay network, my neighbors are the peers I am aware of, *i.e.,* of which I know the address. As a consequence, traditional models can not be used to model peer-to-peer networks.

New Models. Shaker and Reeves [129] gives an intuitive and simple formalization of the bootstrapping problem. Recall that, to be inserted, a node first needs to discover any node already in the system, by using an out-of-band mechanism. Shaker and Reeves [129] put in words the fact that any peer-to-peer system needs a *weakly-connected* bootstrapping system, *i.e.,* able to gather the

addresses of all alive nodes inside the overlay to ensure its convergence to a connected consistent overlay starting from any possibly disconnected topology. In the same paper, they give a self-stabilizing protocol to maintain an overlay network, assuming the presence of a continuous *weakly-connected* bootstrapping service. Each node periodically initiates lookup to the bootstrap system. Dolev and Kat [122] propose a similar *bootstrap-dependent* self-stabilizing overlay network based on their *HyperTree* structure. The HyperTree is a virtual tree structure built on the IP addresses in which in-degree and out-degree of nodes are ensured to be $b \log_b n$ where n is the actual number of machines and b, an integer greater than 1. The maximum number of hops in a lookup in the HyperTree is bounded by $\log_b n$. Following these two works, in [76], a model is proposed for the design of distributed algorithms for large scale systems, opening doors for further systematic investigation of self-stabilization problems in peer-to-peer networks. A spanning tree maintenance protocol illustrates the model. Some recent works focus on the publish/subscribe paradigm, often implemented by peer-to-peer networks. Several papers, *e.g.,* [49, 154] design such protocols, but enhancing them with the self-stabilizing property.

Chapter 3

A Dynamic Prefix Tree for Service Discovery

This chapter[1] initiates the description of our solution, whose core element is a distributed trie. We call our approach *DLPT* for *Distributed Lexicographic Placement Table* or *DPT* for *Dynamic Prefix Tree* as referred to as in several papers, see [43, 82]. We detail the maintenance and the use of our particular compact trie, namely, a *Greatest Common Prefix Tree*, in a distributed environment. We give the detailed algorithms allowing to maintain this structure as services are registered and unregistered. We also give a first intuitive fault-tolerance mechanism based on replication. We discuss how to achieve load-balancing and topology awareness based on the replication scheme. After maintenance, we detail the querying process in such a structure and how to improve its performance by a simple cache mechanism. The evaluation of our approach relies on complexity analysis and intensive simulation.

In the following section, we state how we model services to be inserted in the indexing system. In Section 3.2, we present the distributed structure we use. Then, the different parts of the maintenance of such a tree in a message passing environment are detailed in Section 3.3. Query mechanisms are provided in Section 3.4. Complexities are discussed in Section 3.5. We present and analyze a series of simulation results in Section 3.6. Finally, we give a first comparison with related work and introduce the remainder of the dissertation.

3.1 Modeling Services

As we already mentioned Page 22, users want to do multi-attribute and range searches. Let us consider the example below describing the service S:

$$\texttt{S} = \{\ \texttt{DGEMM}, \texttt{Linux Debian 3}, \texttt{PowerPC G5}, \texttt{node1.cluster2.grid}\ \}$$

This represents a `DGEMM` routine, running under Linux on a PowerPC architecture provided on the server whose address is `node1.cluster2.grid`. To allow the retrieval of S according to each of its attributes, a $(key, value)$ pair is created for each of them, the value being the information needed

[1] The work presented in this chapter has been published in an international journal [CDT07], an international conference [CDT06] and a national conference [Ted06]. Publications and research reports are available at http://graal.ens-lyon.fr/~ctedesch/research.html.

to connect the service, *i.e.,* its address:

```
(DGEMM, node1.cluster2.grid)
(Linux Debian 3, node1.cluster2.grid)
(PowerPC G5, node1.cluster2.grid)
```

As one of the possibility we want to provide is automatic completion of partial strings, we need an indexing structure reflecting the *lexicography* of keys. *Tries* provide a simple and efficient way to store keys according to their lexicography or *prefix relation*. They have several good properties we already briefly mentioned in the previous chapter (Page 34) and which we will detail further in our particular case. We use a particular compact type of tries, called a *greatest common prefix tree*.

3.2 Distributed Structures

We now focus on the description of the structure we use to maintain the information of services available in the platforms. As already described in Chapter 2, a *trie* is a particular kind of tree for storing strings in which there is one node for every common prefix. For instance, each node of a *binary* trie is labeled by a prefix that is defined recursively: given a node with label l, its left and right child nodes are labeled $l0$ and $l1$, respectively. Figure 3.1 shows a trie built on some names picked within the set of routines of the BLAS library (`DGEMM`, `DGEMV`, `DTRSM`, `DTRMM`, `DSYRK` and `DSYR2K`). *PATRICIA tries* [106], also called *radix trees* when storing integers, are compact versions of tries. The vertices (or edges between vertices, the lexicography being expressed the same way) of a PATRICIA trie are labeled with sequences of characters rather than with single characters, keeping a vertex only if necessary and also reducing depth and size of the structure. Figure 3.2 gives the compact version of the trie in Figure 3.1 for the same set of keys.

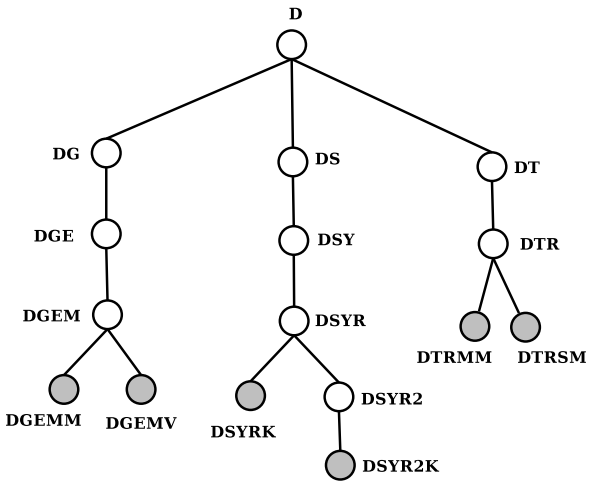


Figure 3.1: A trie.

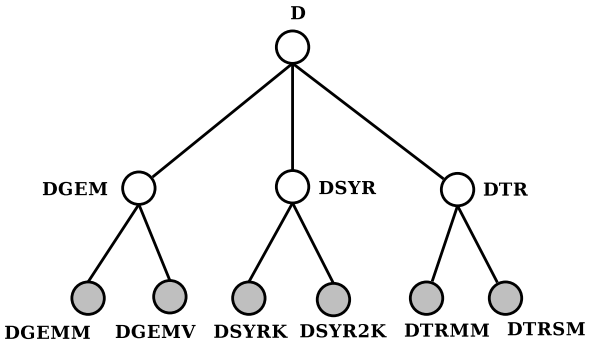


Figure 3.2: A compact *PATRICIA* trie.

The distributed structure we rely on is very similar to a PATRICIA Trie. But, for the sake of clarity and to avoid misunderstandings, we now describe the very structure we use and which we call a *Proper Greatest Common Prefix Tree*.

Proper Greatest Common Prefix Tree. We now formally describe the distributed structure we maintain. Let an ordered alphabet A be a finite set of letters. Denote $\prec$ an order on A. A non empty *word* w over A is a finite sequence of letters $a_1, \ldots, a_i, \ldots, a_l$, $l > 0$. The *concatenation* of two words u and v, denoted as $u \circ v$, or simply as uv, is equal to the word $a_1, \ldots, a_i, \ldots, a_k, b_1, \ldots, b_j, \ldots, b_l$ such that $u = a_1, \ldots, a_i, \ldots, a_k$ and $v = b_1, \ldots, b_j, \ldots, b_l$. Let ϵ be the *empty word* such that for every word w, $w\epsilon = \epsilon w = w$. The *length* of a word w, denoted by $|w|$, is equal to the number of letters of w—$|\epsilon| = 0$. A word u is a *prefix* (respectively, *proper prefix*) of a word v if there exists a word w such that $v = uw$ (resp., $v = uw$ and $u \neq v$). The *Greatest Common Prefix* (resp., *Proper Greatest Common Prefix*) of a collection of words $w_1, w_2, \ldots, w_i, \ldots$ ($i \geq 2$), denoted $GCP(w_1, w_2, \ldots, w_i, \ldots)$ (resp. $PGCP(w_1, w_2, \ldots, w_i, \ldots)$), is the longest prefix u shared by all of them (resp., such that $\forall i \geq 1, u \neq w_i$).

Definition 1 (PGCP Tree)**.** *A* Proper Greatest Common Prefix Tree *is a labeled rooted tree such that both following properties are true for every node of the tree:*

1. *The node label is a proper prefix of any label in its subtree;*

2. *The greatest common prefix of any pair of labels of children of a given node is the same and is equal to the node label.*

We now discuss a bit this definition with the following remark:

Remark 1. *According to Definition 1, in a PGCP Tree, each node has a unique label.*

Proof. Assume by contradiction that there exists two nodes p_1 and p_2 in the tree having the same label l. It is important to use the adjective *proper* with care. Note that the second property of Definition 1 does not use *proper* greatest common prefixes, in which case the remark could not be ensured, but just *greatest common prefix*. There are two cases to consider:

1. p_1 and p_2 have the same parent, in which case, according to the second property, the label of the parent must be labeled with this same label again, which can not be true because of the first property.

2. p_1 and p_2 have two distinct parents but they have a common ancestor a, root of smallest subtree containing them both. Remark that in the set of children of a we can find exactly two nodes a_1 and a_2 also ancestors of p_1 and p_2 respectively. Then a_1 and a_2 both prefixes l and thus either they are equal, and this is similar to Case 1 (p_1, p_2 being replaced by a_1 and a_2), or one prefixes the other (let's say a_1 prefixes a_2, without loss of generality), which is also impossible between siblings because it would mean that the greatest common prefix of the labels of a_1 and a_2 is the label of both a_1 and a, which is impossible because a properly prefixes all labels in its subtree, according to the first property.

□

We now detail the distributed construction and maintenance of this structure as services are registered and unregistered by servers, *i.e.,* as strings are added to and removed from the tree.

3.3 Maintaining a Logical PGCP Tree.

Each node is identified by one given key. We consider two types of keys.

Real Key. A node identified by a **real key** stores the reference of at least one service. For instance, `DGEMM` is considered as a real key as soon as a server has declared a service under the `DGEMM` name. Note that the leaves of the tree are always identified by real keys.

Virtual Key. A node identified by a **virtual key** is the root of a subtree in which nodes' labels share this virtual key as a common prefix. These nodes are created to reflect the common prefixes shared by keys and in order for the tree to satisfy Definition 1.

Figure 3.3 shows the construction of such a tree, when three services are declared sequentially.

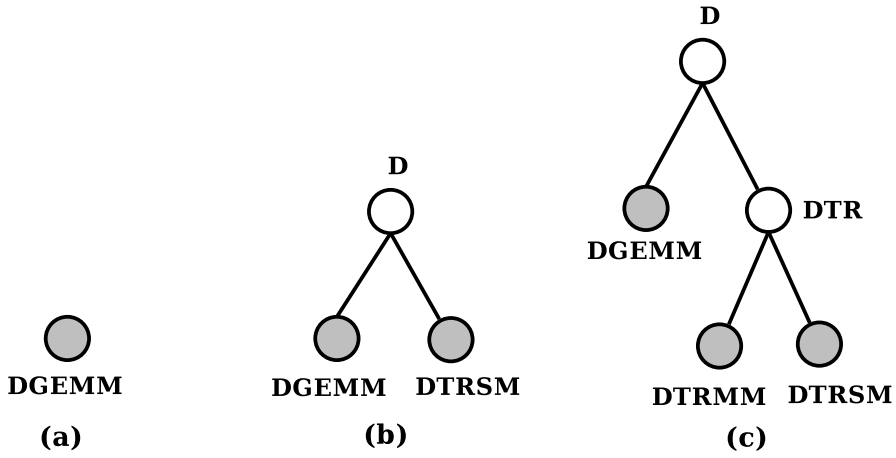


Figure 3.3: Construction of a proper greatest common prefix tree. Nodes storing some services' references (labeled by real keys) are grey-filled, the others are labeled by virtual keys. (a) First a `DGEMM` is declared. (b) A `DTRSM` is declared resulting in the creation of their parent, whose label is their greatest common prefix, *i.e.,* `D`. (c) Finally, a `DTRMM` is declared and the node `DTR` is created.

The Mapping Problem. Recall that the tree we maintain is a logical structure that needs to be mapped on the physical network. A node of the tree is a *logical node*. A processor of the underlying physical network is called a *peer*, and a logical node is called simply *node*, henceforth. A node is an active process running on one peer. In other words, each peer of the physical network hosts a part of the tree, *i.e.,* runs some processes being nodes in the tree. The system behind the cloud in Figure 1.1 of Chapter 1, Page 18 is now a bit less dark, as illustrated on Figure 3.4. How nodes of the tree are mapped onto the peers of the network is here out of topic, we will focus on that issue and detail how we handle the mapping procedure in the next chapter. In the remainder of this chapter, we assume a GetNewProcess() primitive that starts a process (waiting for a proper initialization) on a randomly picked peer among the set of available peers. This primitive relies on any system able to return the address of an available peer, *e.g.,* a DHT structuring the underlying network, or a centralized repository of nodes.

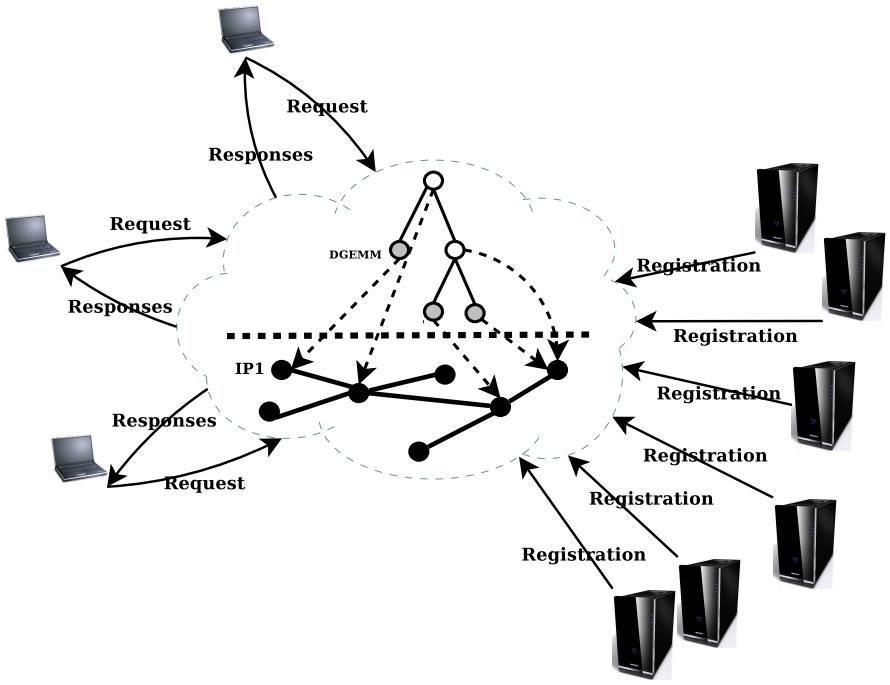


Figure 3.4: Behind the service discovery cloud: an indexing virtual tree (grey-filled and transparent nodes) built as services are declared is mapped on the (black-filled) peers of the physical network.

3.3.1 The Insertion Algorithm

We now assume the initial (possibly empty) tree satisfies Definition 1. When registering a new service, the server providing it contacts a random peer it knows by an out-of-band mechanism (web page, name server, ...) and sends an insertion request for its service represented by the pair `(key = service_name, value = server_IP)`. It is important to note that this contact peer can be any node in the tree, not necessarily the root. The contact peer hosts a set of nodes of the tree and thus has at least one access point to the tree. The request is then routed according to the key.

We now provide the algorithm to insert a service in the PGCP tree using message passing. The detailed distributed process is given by Algorithm 1, Page 61, on node p (p is the unique identifier of the node). p has a label l_p, a parent node denoted f_p whose label is l_{f_p}, a set of children C_p, which is a set of pairs $\{(id, label)\}$, each pair representing one child. For the sake of clarity, we will use the notation $q \in C_p$ to refer only to the identifier of a node in C_p. Each node p stores a set of values (IPs of servers) denoted δ_p whose common key (name of the service) is equal to l_p. The algorithm relies on three other primitives.

- PREFIXES(k) returns the set of words properly prefixing k. For instance, PREFIXES(10101) returns $\{\epsilon, 1, 10, 101, 1010\}$, where ϵ is the empty string.
- GCP(k_1, k_2) returns the greatest common prefix shared by k_1 and k_2. For instance, GCP$(101, 100) = 10$.
- INITPROCESS($lbl, values, parent, parent_{lbl}, children$) initializes a new node in the tree. More precisely, it sets the values on a process recently started by the GETNEWPROCESS() primitive.

Each node, on receipt of an insertion request on the service $s = (k, v)$ pair applies the following routing algorithm, considering four distinct cases:

k is equal to the local node identifier (Line 2.02). In this case, k is already in the tree. No node needs to be added, and no other node is labeled k since each label is unique. p adds v into its table of values.

k is prefixed by the local node identifier (Lines 2.03-2.10). In this case, p is the root of a subtree that may contain a node labeled k. There are two cases to consider:

1. If p has a child also prefixing k, it is also the root of a refined tree in which k must be inserted, so the request is forwarded to it (refer to Lines 2.04-2.05.)
2. Otherwise, no node in the tree prefixes k more than p itself and k must be inserted on a child of p which does not exist yet. A new node labeled k is created and v inserted in its table (refer to Lines 2.06-2.10.)

The local node identifier is prefixed by k (Lines 2.11-2.24). In this case, there are again two cases to consider:

1. If p is the current root of the tree, a new node labeled k must be created, as the new root of the tree and parent of p. This is done by Lines 2.12-2.15.
2. If p has a parent, either this parent is again prefixed by k and k must be moved upward (see Lines 2.17-2.18), or it is not the case, and the only possible location of a node storing the service s is between p and its parent. Such a logical node is created in Lines 2.19-2.24.

No prefix relation (Lines 2.25-2.38). Even if k and l_p have no prefix relation, the process is similar. If p is not the root of the tree and if the label of its parent node is equal to or prefixed by the common prefix of k and l_p, the request must be moved upward in the tree to be sure to reach the root of the subtree potentially storing k. (see Lines 2.26-2.27). Otherwise (starting from Line 2.28), k cannot be anywhere else in the tree, and s must be stored on a sibling of p. Obviously, the common parent of the node labeled by k and the node p does not exist. Remember the steps (b) and (c) of Figure 3.3: two nodes are created, one labeled k and one labeled by the greatest common prefix of k and l_p (possibly the empty string).

In order to initiate a node/process, we need to send the values of the newly created node to the process previously obtained using the GetNewProcess() function, this is achieved using a couple of synchronized messages: (i) a Host message is sent by the initiator of the creation of the node. Upon receipt, a node initiates its variables (see Lines 3.01-3.03) and sends back an *acknowledgment* through the HostDone message. When a new node is created, we need to inform its parent that it has a new child. Also, in the last of the four previous cases, one child is removed and replaced by a new node (this case is illustrated on Figure 3.3, Page 46, when the node `D` replaces its child `DTRSM` by the newly created node `DTR`). This update is handled by a simple set of messages, namely RemoveChild and AddChild whose receipt is detailed by Lines 4.01 and 4.02.

3.3.2 Service Removal

If some server, for some reason, does not want to provide a given service $s = (k, v)$ anymore, it informs the system of this removal by sending a *deletion request* on s to any node in the tree. On receipt, the request is routed according to k similarly as before and reaches the node labeled k, storing the value v among other values for the service labeled k. On receipt of the request, this node then removes v from its table.

In the case where v was the last remaining value of services labeled k, the node may not be useful anymore and removed from the tree. Deleting a node is done if both following requirements are reached:

1. no values is stored on the node anymore,
2. the node has no children.

If the case a node enters this configuration, it sends a message to its parent to inform it of its imminent deletion and stop itself. If the node is the root of the tree, has no parent to inform, this is the last remaining node and the tree ends.

A service can also be updated with new information. We do not detail the update process because it is very similar to the insertion and removal processes, except that no node is created or deleted. Once the node storing the information of the service to be updated is reached (the node always exists since the service has been previously declared by the server), nothing is done except updating the information.

3.3.3 Replication

To face the dynamic nature of the underlying network and ensure the consistency of the routing while preventing data loss, we first proposed a replication scheme. Assume a global replication factor k, which denotes the number of distinct peers on which each logical node must be present. Such a replicated trie is shown in Figure 3.5 with $k = 2$. Obviously, k-replication assumes that less than k crashes occur between two replication processes. Otherwise, only probabilistic results could be given on the availability of the system.

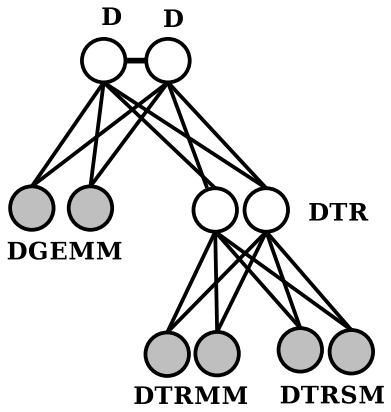


Figure 3.5: Example of a replicated PGCP tree (2-replication).

The replication mechanism, whose detailed pseudo-code is given by Algorithm 2, Page 62, is started periodically on the top of the tree. As the nodes are replicated, each node maintains between 1 and k links for each of its neighbors, each neighbor being located on 1 to k distinct peers. As a consequence, each node p should maintain a set of references of process C_{lbl}, for each child's label lbl. Similarly, each node now maintains k connections to its parent (in the variable F). Each node finally may need to know its own replicas, denoted R_p throughout the process.

A simple leader election (Lines 6.02-6.04) prevents the replication process to be triggered concurrently by several roots (in the case the top of the tree was already replicated). Each node periodically scans its replicas to have knowledge of the remaining alive roots. If p is the minimum identifier among the replicas of the root, p triggers the replication, beginning by ensuring the k-replication of the root (see Lines 6.05-6.10). The elected replica obtains new processes and launches new roots to reach k roots, by sending them the information on p itself through the HOST message. On receipt, as detailed by Lines 9.01-9.06, the new process starts itself as a root. This replication is synchronous, to avoid to communicate with non-initialized processes. Once the root is replicated, the replication of the tree is started by the REPLICSUBTREE message sent by the elected root to itself (Line 6.11). As we will detail, the replication is a protocol moving downward, each node replicating its children and triggering the replication of the subtrees of its own children by the REPLICMESSAGE. As only one replica can replicate its children at a time, the chosen replica needs to inform its replicas of their new children (newly created replicas of their child nodes). For this reason, the REPLICMESSAGE contains the set of other replicas of the chosen one to replicate its subtree.

On receipt of the REPLICNODE message along with the set of references of its replicas R, a node starts replicating its subtree, processing each child sequentially (Lines 7.01-7.14). For each child dc, it scans the available replicas through the GETALIVEREPLICAS() function to have the knowledge of alive replicas for dc. It requests one alive child r (recall that at least one should be alive) to replicate itself on new processes (obtained Lines 7.05-7.07) with a synchronous set of communications (Lines 7.08-7.10 and 8.01-8.06). Finally, one replica b is chosen to continue the replication in the subtree of dc. The GETBESTREPLICA() function executes a set of measurements on the replicas to know which one *locally* optimizes a given metric (typically the latency). The replication is launched in the subtree of dc by *delegation* to b, and continues sequentially with other children of p.

We assume that no message is lost and that processes whose reference is returned by the GETNEWPROCESS(), GETRANDOMNODE(), GETALIVEROOTS() or GETBESTREPLICA() functions remains alive until the replication job is done.

We now discuss two advantages coming along with this replication process and that can be used to improve the performance of the prefix tree.

Load balancing

Since each node is replicated k times, each node has k choices to route a single request. A straightforward load balance mechanism can consist in a *round robin* selection of the replica to which the request is forwarded, thus uniformly distributing requests for one node on its different replicas.

Topology Awareness

An example of the replication process is given in Figure 3.6. As highlighted on this figure, each locally chosen *best replica* is part of a spanning tree, built by local measurements. By keeping this replica for each child, we can use the best local connections to forward requests, thus easily taking

into account the performance of the underlying network without need for any external tool and without any extra cost.

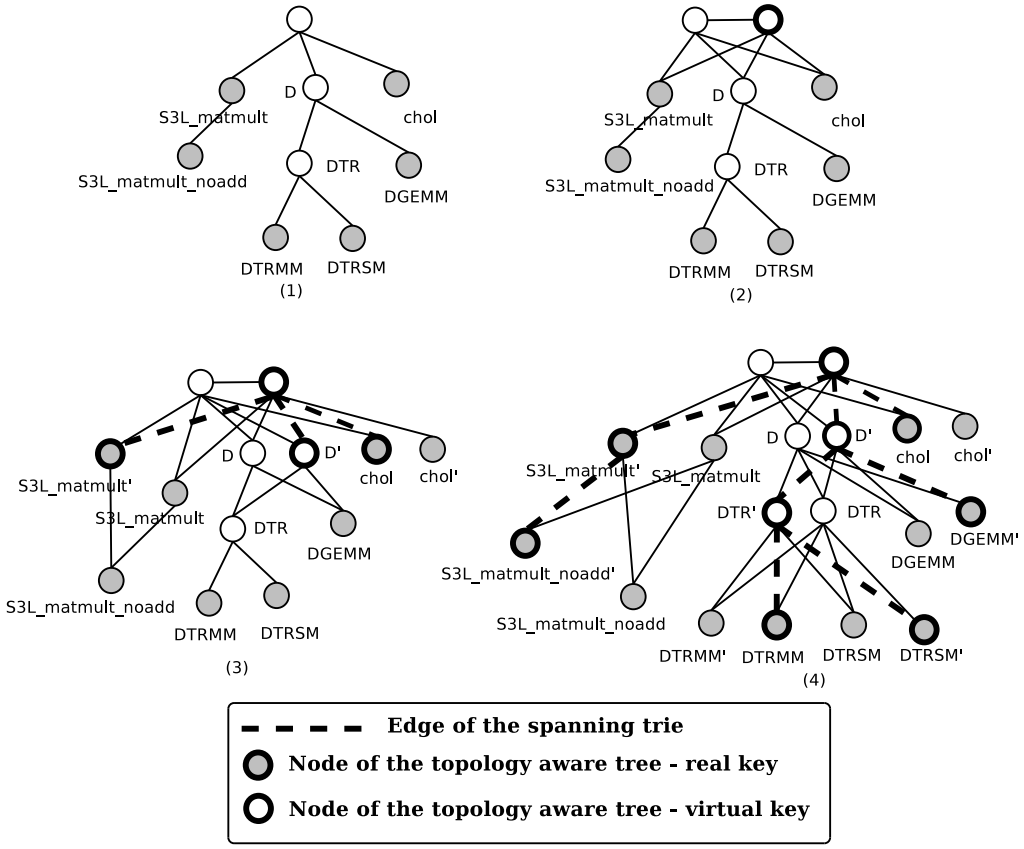


Figure 3.6: Replication and topology awareness.

Note that these two optimizations are incompatible since in one case, we select only one peer to route and in the other case, we try to send the requests uniformly between several peers. Afterward, the purpose is to find the right trade-off between the two objectives.

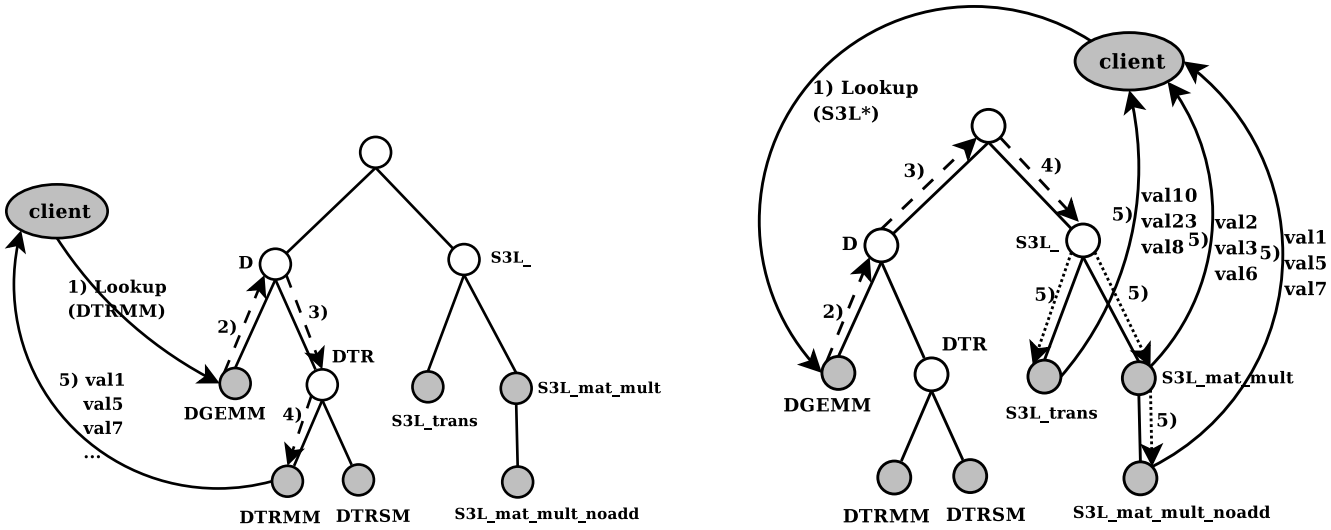
3.4 Querying the PGCP Tree

We now describe the detail mechanisms allowing the service discovery itself. We distinct two general kinds of queries: exact-match queries, on one particular key, and range queries.

3.4.1 Exact Match

Processing a basic exact-match query is based on an algorithm similar to the service insertion previously detailed. This process is illustrated by Figure 3.7(a). The client issues a request and sends it to a node whose reference has been obtained by the out-of-band mechanism. The request is routed in the tree by a simple top-down traversal similar to the previous insertion algorithm (moving upward until finding the root of the subtree that may contain the node with the requested label and then

moving downward). Finally, if such a node exists, it is eventually reached and it sends back the values of the service wanted to the client.



a. Exact-match b. Automatic completion

Figure 3.7: Processing a discovery request. The client sends the request to a node it knows (1). The request is routed (2,3,4). Responses are sent back to the client from the node storing the key or from the nodes in the subtree whose root is the *responsible* node for this request (5).

3.4.2 Range Queries

Supporting automatic completion of partial search strings is achieved in a two phases distributed process, as illustrated on Figure 3.7(b), where a client seeks services of the SUN S3L library [6] and issues the query `S3L*`.

1. We first consider the explicit part of the search string, here `S3L`. The request is thus routed according to `S3L`, similarly as with exact-match queries, except that the destination node is the node whose identifier is the smallest key **equal or prefixed by** `S3L`. On Figure 3.7(b), this node is labeled `S3L_`. The requested keys are in the subtree rooted at this node.

2. Once the root of the subtree pertained by the partial string is found, it remains to traverse every nodes of the subtree, in parallel. Each node sends its values to the client. The client can stop the reception, if satisfied with the current set of received values.

Supporting range queries is based on the same process. The two extreme values of the range share a common prefix. Based on that, it remains to execute the first of the two previous phases on this prefix and launch the parallel scan of the subtree, propagating the request to the nodes of each pertained subtree.

3.4.3 Cache Optimizations

We now describe some easy-to-implement cache mechanisms able to drastically reduce the cost of a request.

Cache During Exact-Match Queries

Popular keys result in bottlenecks on peers running processes storing popular keys. To avoid their appearance, we propose to disseminate popular keys on several peers, by caching retrieved values on the way back to the client, by reverse routing (the values are sent to the client **and** cached on nodes on the way back to the contact node of the query). If a request reaches a peer caching values corresponding to a recently requested key (and this is probable in the case of popular keys), the routing ends and a response is immediately sent to the client, also distributing the work load.

Cache During Partial String Searches

The traversal of the subtree pertained by the range is as expensive as the subtree is large, since it requires $O(n)$ messages, n being the number of nodes in the subtree. A simple optimization consists in caching the values on the root of the subtree, allowing future requests to be (at least partly) satisfied without the need to wait for the results of the complete traversal.

3.4.4 Multi-Attribute Searches

As illustrated on Figure 3.8, supporting multi-attribute queries is achieved using a simple extension of the previous algorithms to a multi-dimensional system in which each dimension (or *type* of attribute is maintained by a distinct overlay. Then, the value/address of a service having several attributes is stored in each overlay. For multi-attribute requests, like {`DTRSM, Linux*, PowerPC*`}, the client sends three independent requests. The request on `DTRSM` will be sent to the *services' names* tree, `Linux*` to the *system tree* and `PowerPC*` to the *processors* tree. Requests are independently processed within each tree and the client asynchronously receives the values and finally intersects the sets of locations obtained to only keep answers matching the three values requested.

3.5 Complexity Discussion

We now discuss complexities of the distributed PGCP Tree. Most of these results are similar to those of any trie complexity studies. Let us consider a non-replicated PGCP Tree T. Characters composing the keys are part of an alphabet $\mathcal{A}$. We assume the cost of the GetNewProcess() as constant. Even if the implementation of this function is out of scope of this chapter, we can assume the underlying system finding new processes to be coupled with a cache system making the start of a new process available at any time by having some references of alive peers in cache, thus justifying the *constant* time cost of getting a new process.

Routing Complexity. If we assume the length of keys in T bounded by L, then the routing complexity is $O(L)$. To show that, first note that, based on the fact that, according to Definition 1, a child is properly prefixed by its parent, the path from any node up to the root can not exceed L. Then remark that the routing in the tree always starts by moving upward in the tree and then going downward until reaching the location of the insertion. It remains to see that the local routing decision, *i.e.,* the number of scans needed in the routing table to choose the next hop is constant (if sending the request to a child, it suffices to look at the first character of the searched key not shared by the label of the local node to find the right child).

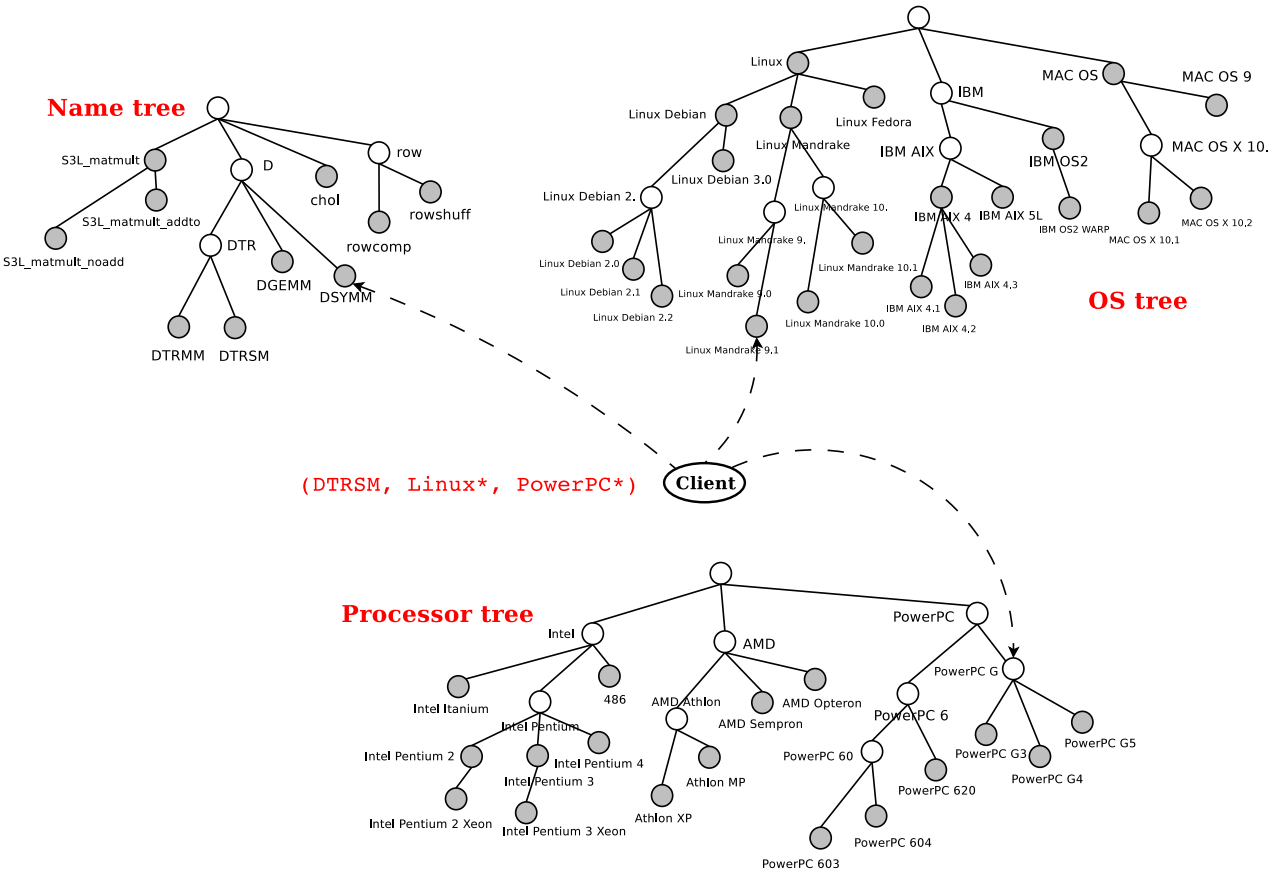


Figure 3.8: Processing a multi-attribute query.

Total Insertion Cost. Note that once the location is found, the insertion itself required a constant number of operations, as we can easily verify looking for Algorithm 1.

Degree. The degree of a node is bounded by the number of characters in $\mathcal{A}$ plus 1. The number of children is bounded by $Card(\mathcal{A})$. To establish that, it suffices to remark that if a node has more than $Card(\mathcal{A})$ children, at least two children share a common prefix longer than their parent label, thus breaking Definition 1. Finally, each node has at most one parent.

Note that, when the tree grows, L tends to be $O(\log N)$: Consider the tree containing all the keys possibly built on $\mathcal{A}$. The number of nodes in this tree is $N = \frac{Card(\mathcal{A})^{L+1}-1}{Card(\mathcal{A})-1} = O(Card(\mathcal{A})^L)$, and $L = O(\log_{Card(\mathcal{A})} N)$.

Range Queries. Based on the previous discussion, we easily find that the latency of a range query is also bounded in time by $2L$. As detailed previously, the range query process is based on two phases, made of (i) routing, (ii) propagation in the subtree. As the second phase runs in parallel, its time is bounded by the time to traverse the path leading to the deepest node of the subtree. The number of messages required is clearly in the size of the subtree.

Replication Complexity. A study of the replication mechanism detailed in Algorithm 2 is enough to determine the complexity of the replication process. Dealing with the root's replication Lines 6.02-6.11, it appears that no loop exceeds k steps, $Card(R_p)$ being by definition lower than k. Then, the k-replication of the set of children of one node achieved by Lines 7.01-7.14 exhibits a complexity in $O(d \times k)$ where d is the degree of the considered node. As this process is repeated for each node in the tree, the total message complexity is in $O(N^2 \times k)$. Dealing with time complexity, we need to remark that each subtree rooted at one different child of one node is replicated in parallel, but these replications are triggered sequentially by the parent of the roots. Without lost of generality, assume that the set of children of a node is sorted. The replication of node p having a cost c_p, the time complexity to reach a leaf ln is

$$\sum_{p \in ExtPath(ln)} c_p$$

where $ExtPath(ln)$ is the set of children of the nodes on the way to the root from ln, which are lower (in terms of labels) than the nodes on the path themselves. In other words, to reach ln, the replication have first sequentially processed all nodes that are children of nodes on the path from the root to ln and all their lower siblings.

3.6 Simulation

A simulator of the tree has been developed in Java. The dynamic creation of the tree, exact match queries and range queries, as well as the caching mechanisms previously detailed, have been implemented. The keys for the simulation are picked inside a set of keys we may find in computational grids:

- 735 routines/functions of computing libraries,
- 129 processors,
- 189 operating systems,
- 3985 fictional addresses of servers (IPs, reverse notations addresses). A user which trusts a given cluster, can specify it in the request. Then, we need one PGCP tree to be maintained on the addresses in reverse notation, as for instance achieved in *in-addr.arpa* schemes. Such a tree is illustrated on Figure 3.9. To automatically complete address keys, the address of a service is reversed to become a key and another way to retrieve services,
- 20000 keys randomly created on the Latin alphabet of size up to 20, to investigate further the scalability.

The random distribution used is the uniform one.

3.6.1 Building the Tree and Insertion Requests

We have first validated the logical algorithm of insertion of a newly declared service. Figure 3.10 shows the evolution of the size of the tree according to the number of insertion requests submitted to the trie, with randomly picked keys. On the left, this evolution is given for the four data sets. As expected, as soon as all keys have been registered, the trie does not grow anymore, all following

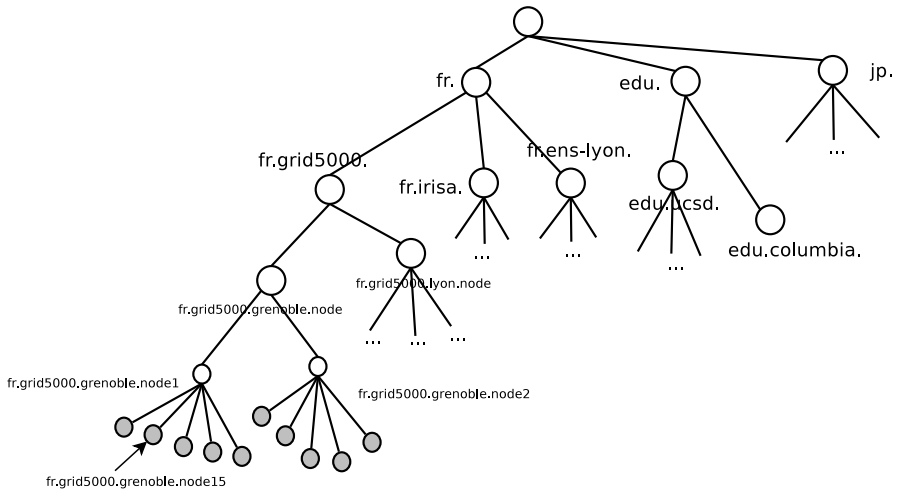


Figure 3.9: A PGCP tree for storing addresses of servers.

insertion requests declaring already picked keys. On the right, we provide the same result but divided by the number of requests. In the same way, this fraction tends towards 0, except for random words, since the probability to randomly build a key already in the tree is very small (20000 keys among $\frac{26^{21}-1}{25} \approx 2 \cdot 10^{28}$ possibilities). In this last case, the tree size remains proportional to the number of insertion requests processed (up to 1.5 times the number of requests.)

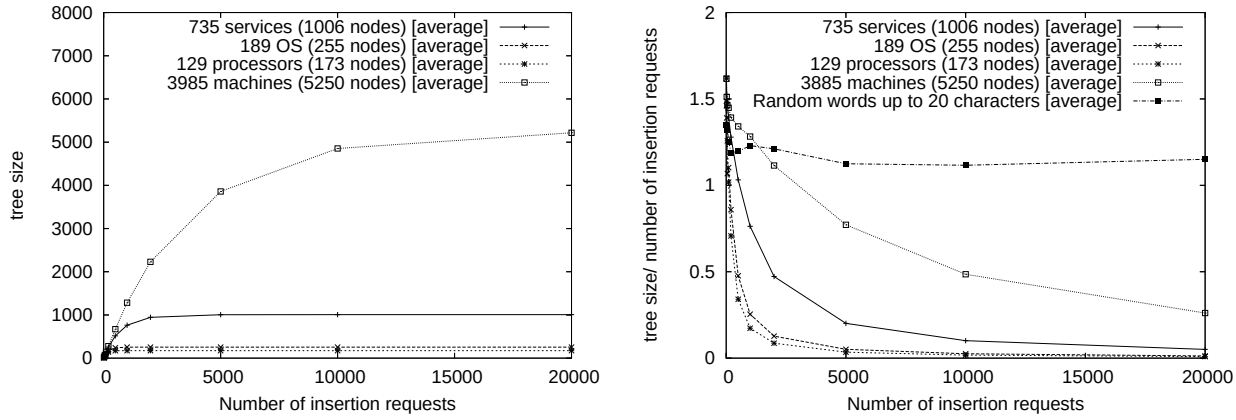


Figure 3.10: Evolution of the tree size according to the number of insertion requests.

Then we have run experiments with insertion requests, to get information on the number of hops needed in the tree to reach a given location in the tree. On the left part of Figure 3.11, we give the number of hops in the case where all the requests enter the trie by the root, to allow an estimation of the average depth of the tree (here, approximately 4 for *grid-oriented* keys). On the right part of Figure 3.11, we show the number of logical hops required to process the request, choosing a random contact node. For these last experiments, a set of 10000 random words has been used. The behavior is close to the one described in Section 3.5: the curve quickly reaches an average depth and then

follows a logarithmic behavior, even for big set of random words.

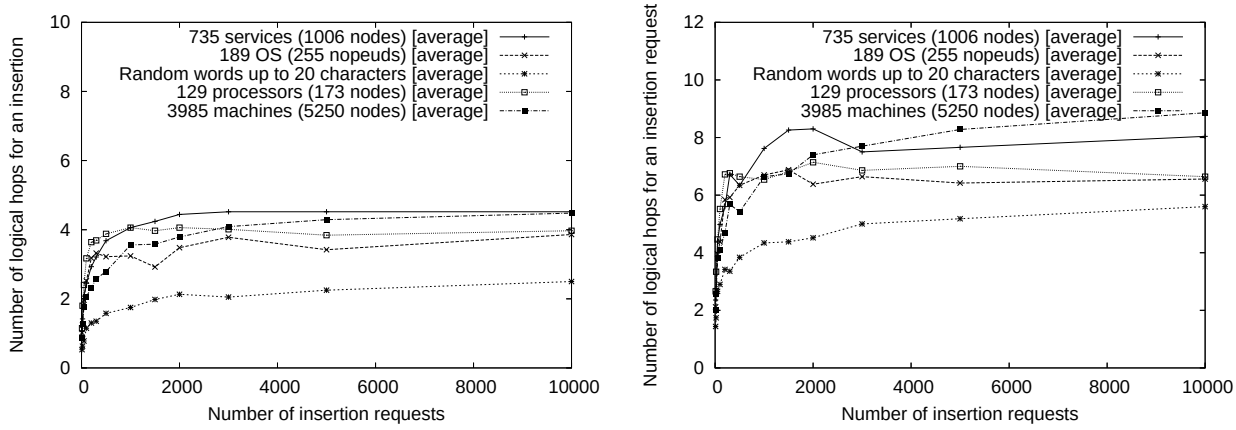


Figure 3.11: Average number of hops (from the root, on the left, and from a randomly picked contact node, on the right).

We have also studied how the tree grows according to the number of distinct declared keys. Each key is now inserted only once. As we see on Figure 3.12, the total number of nodes in the tree (nodes storing virtual keys and nodes storing real keys) is proportional to the number of inserted (real) keys. The sizes of the trees are summarized in the Table 3.1. To sum up the results, the nodes storing virtual keys represents 30% of the tree with a standard deviation of approximately 2.5%.

	Services	Systems	Processors	Machines
Number of real keys	735	189	129	3985
Number of nodes of the trie	1006	255	173	5250
Percentage of virtual keys	29,32%	34,38%	30.93%	27,94%

Table 3.1: Percentage of virtual keys created for each set of keys.

3.6.2 Interrogation Requests and Cache

Then we have studied the number of hops required to route search queries. The results illustrated on Figure 3.13 are similar to those observed previously with insertion requests.

Finally, we studied the gain involved by the cache mechanisms initially developed to avoid the bottlenecks on nodes storing popular keys. Figure 3.14 shows the number of hops for a set of randomly-picked keys among all keys stored in the tree both with and without the caching mechanism. It shows that the number of hops can be significantly reduced. Even with a small cache size (here, 50 values, using a FIFO policy) the number of hops to find values for the requested key significantly decreases, going approximatively from 8 to 5, after a few number of search queries.

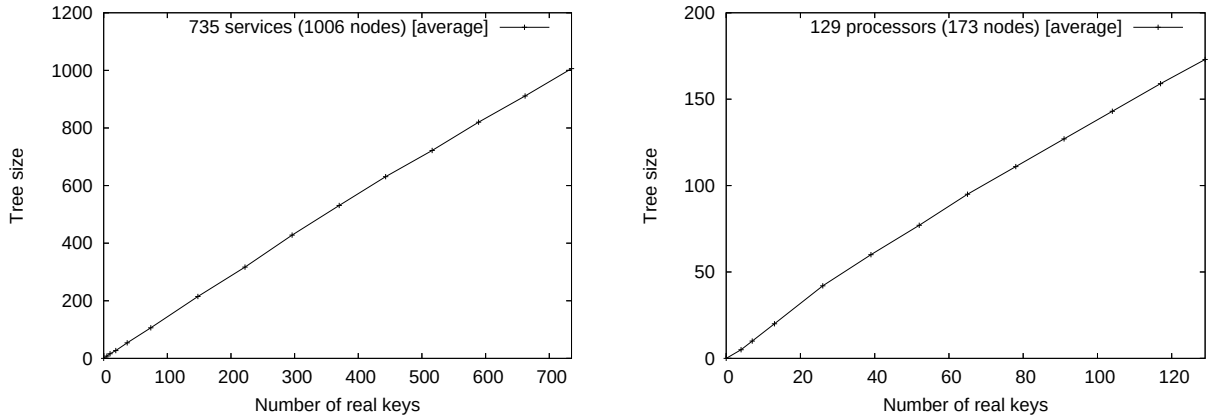


Figure 3.12: Proportionality between the tree size and the number of real keys.

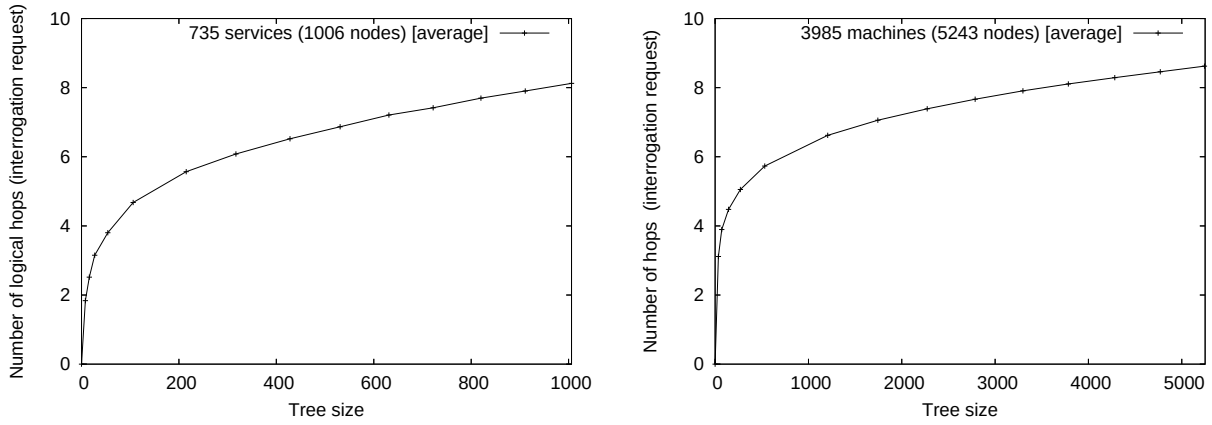


Figure 3.13: Number of logical hops for exact-match queries, according to the tree size.

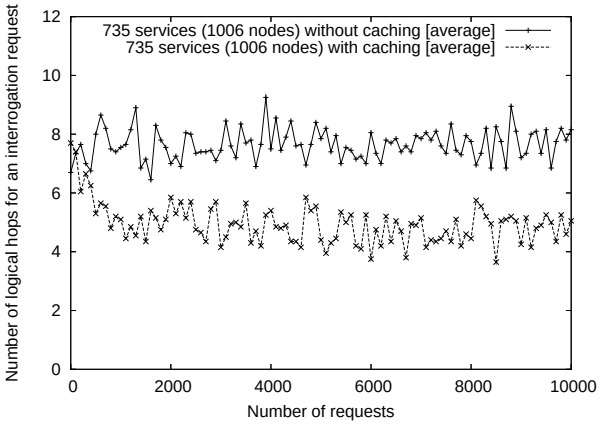


Figure 3.14: Number of hops for exact-match queries with and without the cache mechanism.

3.7 Discussion

3.7.1 PGCP Tree Advantages

As the PGCP tree structure is a compact version of tries, the depth of the tree will be most of the time lower than the worst case L, where L is the maximum length of keys. This behavior has been highlighted by simulation, for instance on Figure 3.11, where in spite of the maximum length of the keys, which is 20, and the average length, which is 10, starting from the root, the average number of hops required is just above 2. Obviously, the *lexicographic* fashion of tries allow to efficiently and easily support automatic-completion and range queries, by knowing exactly where to search. Searches appear to be less costly and more *localized*, than searches in binary search trees (BST) for instance. Moreover, BST and the well known B-tree [28] always require to start the search at the root. Even if the root can be reached starting from any node in BST or B-trees, using our approach allow to reduce the load on the root and to reduce the average number of hops, by moving up to the root only if it is necessary.

3.7.2 Advantages and Drawbacks Relative to P-Grid and PHT

To the best of our knowledge, P-Grid and PHT are our closest related work (trie-based overlay network mapped onto a peer-to-peer network). See Chapter 2, page 36.

First, let us notice that complexities are similar, complexities being bounded by length of keys and the number of possible characters.

P-Grid [48] and PHT [117] does not provide any topology awareness. Our approach proposes hints, as an extension of the replication process to easily take it into account in a greedy fashion.

PHT naturally offers a good fault-tolerance in the sense that one fault does not affect a subtree but only one node. But this is possible only because they have an underlying DHT supporting exact-match queries whatever the state of their trie is. DHT is used as a fallback mechanism if the trie becomes inconsistent.

P-Grid offers a better load-balancing by avoiding the root bottleneck, by building shortcuts, in a way very similar to the Kademlia overlay network [105]. However, the P-Grid good load balance property holds only for homogeneous popularity of keys and homogeneous capacity of underlying

processors, or *peers*. In a more general point of view, assuming heterogeneity of these two parameters requires new algorithms to be designed to take it into account. Moreover, we should take it into account in a dynamic way, since popularity of keys may vary in time. Chapter 4 focuses on the solutions we developed to solve this problem.

Finally, neither P-Grid nor PHT is able to recover from arbitrary failures. These systems completely rely on replication for their fault-tolerance. The replication is costly and never ensure to be sufficient to keep the system safe facing failures, especially if an incorrect configuration (for instance if information about my neighbors was wrongly initialized) is replicated, in which case, the problem may spread over the network. As a direct consequence, the discovery process can not work properly anymore. When the replication becomes inefficient (in case of incorrect configuration, or if the replication failed or had a too small factor), we need to rely on alternative *best-effort* policies able to recover starting after an arbitrary number of failures/errors. Similarly, the algorithms presented in this chapter are not *self-stabilizing*, *i.e.,* able to recover if transient failures occur. As we explained in the previous chapter, self-stabilization is a promising alternative to address fault-tolerance in P2P systems. In Chapter 5, we investigate the power of self-stabilization for fault-tolerant peer-to-peer trie-structured systems by presenting the self-stabilizing algorithms we developed, able to repair our architecture after crashes.

3.8 Conclusion

In this Chapter, we have presented the initial design of our system for service discovery in computational grids. Our architecture is based on a PGCP tree, dynamically constructed as servers register their services. We focused on the *logical* part of the architecture, leaving mapping considerations aside. We provided the complete message passing algorithms to maintain such an architecture. A first attempt at injecting fault-tolerance into our architecture has been given, based on a complete replication of the tree. We gave some hints on the possibilities raised related to topology awareness and load balancing in this kind of trees. Some simple cache mechanisms were described, allowing another way to balance load and improve the response time. We discussed the complexities of the construction and the use of the tree and gave some simulation results showing the relevance and the efficiency of using such a tree for the service discovery in this context. We finally discussed the advantages and drawbacks of our tree relative to the similar architectures and our closest related work. As discussed above, Chapter 4 presents a self-contained approach for the problem of mapping the tree efficiently on the network without the need for an underlying system. Chapter 5 presents the self-stabilizing approaches able to repair our tree after arbitrary transient failures.

Algorithm 1 Service insertion, on node p

1.01 **Variables:** l_p, the label of p; δ_p, set of service values stored on p
f_p, the parent of p; l_{f_p}, the label of the parent of p
C_p, set of pairs $(identifier, label)$ of children of p

2.01 **upon** receipt **of** <ServiceInsertion, $s = (k, v)$> **do**
2.02 **if** $k = l_p$ **then** $\delta_p := \delta_p \cup \{v\}$
2.03 **elseif** $l_p \in$ Prefixes (k) **then**
2.04 **if** $\exists q \in C_p$: $|\text{GCP}(k, l_q)| > |\text{GCP}(k, l_p)|$ **then**
2.05 **send** <ServiceInsertion, s> **to** q
2.06 **else**
2.07 $n :=$GetNewProcess()
2.08 **send** <Host, $(k, \{v\}, p, l_p, \emptyset)$> **to** n
2.09 **receive** <HostDone> **from** n
2.10 $C_p := C_p \cup \{(n, k)\}$
2.11 **elseif** $k \in$ Prefixes (l_p) **then**
2.12 **if** $(f_p = \perp)$ **then**
2.13 $n :=$GetNewProcess()
2.14 **send** <Host, $(k, \{v\}, \perp, \perp, \{(p, l_p)\})$> **to** n
2.15 **receive** <HostDone> **from** n; $f_p := n$; $l_{f_p} := k$
2.16 **else**
2.17 **if** $k \in$ Prefixes(l_{f_p}) **then**
2.18 **send** <ServiceInsertion, s> **to** f_p
2.19 **else**
2.20 $n :=$GetNewProcess()
2.21 **send** <Host, $(k, \{v\}, f_p, l_{f_p}, \{(p, l_p)\})$> **to** n
2.22 **receive** <HostDone> **from** n
2.23 **send** <RemoveChild, (p, l_p)> **to** f_p
2.24 **send** <AddChild, (n, k)> **to** f_p; $f_p := n$; $l_{f_p} := k$
2.25 **else**
2.26 **if** $(f_p \neq \perp) \wedge (|\text{GCP}(k, l_p)| = |\text{GCP}(k, l_{f_p})|)$ **then**
2.27 **send** <ServiceInsertion, s> **to** f_p
2.28 **else**
2.29 $np :=$GetNewProcess()
2.30 **send** <Host, $(\text{GCP}(k, l_p), \emptyset, f_p, l_{f_p}, \{(p, l_p)\})$> **to** np
2.31 **receive** <HostDone> **from** np
2.32 **if** $(f_p \neq \perp)$ **then**
2.33 **send** <RemoveChild, (p, l_p)> **to** f_p
2.34 **send** <AddChild, $(np, \text{GCP}(l_p, k))$> **to** f_p
2.35 $nc :=$GetNewProcess()
2.36 **send** <Host, $(k, \{v\}, np, \text{GCP}(k, l_p), \emptyset, \emptyset)$> **to** nc
2.37 **receive** <HostDone> **from** nc
2.38 **send** <AddChild, (nc, k)> **to** np; $f_p := np$; $l_{f_p} :=\text{GCP}(k, l_p)$

3.01 **upon** receipt **of** <Host, $(lbl, values, parent, parent\_lbl, children)$> **from** q **do**
3.02 InitProcess $(l_p := lbl, \delta_p := values, f_p := parent, l_{f_p} := parent\_lbl, C_p := children)$
3.03 **send** <HostDone> **to** q

4.01 **upon** receipt **of** <RemoveChild, q> **do** $C_p := C_p \setminus \{q\}$
4.02 **upon** receipt **of** <AddChild, q> **do** $C_p := C_p \cup \{q\}$

Algorithm 2 Replication, on node p

```
5.01  Variables:  l_p, the label of p δ_p, set of service values stored on p
                  F_p, the set of parents of p
                  l_{F_p}, the label of the parent of p
                  C_p, set of pairs (lbl, C_lbl) of children of p where C_lbl is the set of children labeled lbl
                  R_p, set of my replicas

6.01  Periodically:
6.02  if F_p = ∅ then
6.03      R_p := GetAliveReplicas(R_p)
6.04      if min(R_p ∪ {p}) = p then
6.05          while Card(R_p ∪ {p}) < k do
6.06              n := GetNewProcess()
6.07              R_p := R_p ∪ {n}
6.08          for all q ∈ R_p do
6.09              send <Host, (l_p, δ_p, F_p, l_{F_p}, C_p, R_p ∪ {p} \ {q})> to q
6.10              receive <HostDone> from q
6.11      send <ReplicSubtree, (R_p)> to p

7.01  upon receipt of <ReplicSubtree, (R)> do
7.02      R_p := R
7.03      for dc = (lbl, C_lbl) ∈ C_p do
7.04          C_lbl_tmp := GetAliveReplicas(C_lbl)
7.05          while Card(C_lbl) < k do
7.06              n := GetNewProcess()
7.07              C_lbl := C_lbl ∪ {n}
7.08          r := GetRandomReplica(C_lbl_tmp)
7.09          send <ReplicNode, (R_p, C_lbl \ {r})> to r
7.10          receive <ReplicDone> from r
7.11          for q ∈ R_p do
7.12              send <UpdateChildren, (lbl, C_lbl)> to q
7.13          b := GetBestReplica(C_lbl)
7.14          send <ReplicSubtree, C_lbl \ {b}> to b

8.01  upon receipt of <ReplicNode, (F, R)> from f do
8.02      F_p := F
8.03      for q ∈ R do
8.04          send <Host, (l_p, δ_p, l_{F_p}, F_p, C_p, ∅)> to q
8.05          receive <HostDone> from q
8.06      send <ReplicDone> to f

9.01  upon receipt of <Host, (l, δ, l_F, F, C, R)> from f do
9.02      if already running
9.03          UpdateProcess(l_p := l, δ_p := δ, l_{F_p} := l_F, F_p := F, C_p := C, R_p := R)
9.04      else
9.05          InitProcess(l_p := l, δ_p := δ, l_{F_p} := l_F, F_p := F, C_p := C, R_p := R)
9.06      send <HostDone> to f
```

Chapter 4

Mapping and Load Balancing

In the previous chapter, we introduced the logical part of our architecture. The problem of mapping the logical entities on the physical processors (peers) was left aside, assuming the presence of a device (decentralized, like a DHT or centralized, for instance a central server) able to return the reference of a random chosen peer of the network on which a new process was created.

In this chapter[1], we tackle the problem of mapping a logical PGCP tree on a set of peers. Precisely, we address two drawbacks of the design of the system presented in the previous chapter:

1. The presence of an underlying system like a DHT, leads to the need to maintain two layers (one DHT, and one tree over the DHT), making the maintenance cost of this two-layer architecture extremely high.

2. The routing scheme presented in the previous chapter, as well as the heterogeneity on both popularity of keys and capacity of peers could lead to an unbalanced distribution of the load and thus create bottlenecks on different peers.

Tackling these two drawbacks, we here provide two main contributions:

- the first contribution of this chapter is the avoidance of the DHT: we present a self-contained tree overlay network able to maintain a tree storing the information on services available **and** mapping it on the peers.

- The second contribution is the design of a new load balancing heuristic based on local maximization of the *throughput i.e.,* the number of requests processed by the service discovery system. This heuristic is inspired from existing approaches for the load balancing within distributed hash tables (see details in Chapter 2). We enhanced our overlay with this heuristic and another heuristic for DHTs based on the K-choices paradigm, which is, to the best of our knowledge, the only purely-decentralized one that also assumes heterogeneity on both capacity of peers and popularity of keys, in a dynamic fashion.

The rest of the chapter is organized as follow. In Section 4.1, we give the preliminaries of our architecture. In Section 4.2, the detailed algorithms to build, use, and perform some efficient load balancing within our overlay are presented. Section 4.3 provides the results of our simulations, showing the liveness of the architecture and the performance of our heuristic. Finally, before concluding, we give clues for a fair comparison with our related work in Section 4.4.

[1]The work presented in this chapter has been published in [CDT08].

4.1 Preliminaries

4.1.1 System Model

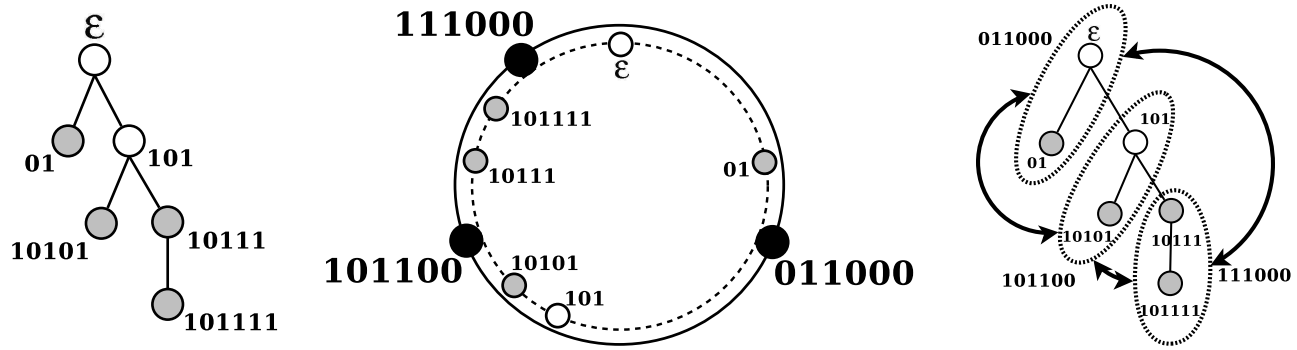
We assume a P2P network made of a set of asynchronous *physical* nodes with distinct IDs. Recall that we use the term *peer* to refer to this kind of nodes. The peers communicate by exchanging messages. Any peer P_1 can communicate with another peer P_2 provided P_1 knows the ID of P_2. Each peer maintains one or more *logical* nodes of a distributed *logical* PGCP tree, each node of the tree being uniquely labeled. As before, we use the term *node* to refer to the nodes of the tree.

4.1.2 Architecture

In the previous chapter, the mapping was achieved through a DHT, in which each peer had a unique identifier, result of the hashing function on the IP address of the peer. The nodes were randomly mapped by hashing their labels. From now on, we consider that IP addresses of peers are still hashed, but, the hash function to find the peer hosting one node is locality-preserving. Seen differently, we can consider that nodes' labels are not hashed but nodes just *placed* on the peer whose ID is the smallest higher than the label of the node, assuming that we hash IP addresses of peers by using a hash function which has values within the set of possible labels of nodes. As a simple example, consider the binary PGCP tree in Figure 4.1(a). The mapping achieved is similar to Chord in the sense that each node is run by the *successor* peer of the node's label.

Have a look at Figure 4.1(b): The tree nodes (transparent and grey-filled on the internal dashed circle) and the peers (big and black filled on the external circle) are in the same *circle* identifier space. Now, each peer is supposed to run nodes whose labels fall in the range between itself and its predecessor peer. For instance, peer `011000` runs nodes 01 and ϵ. If nodes are again connected through tree links (that do not appear on the figure 4.1(b) for clarity), peers are also connected, on a basic ring.

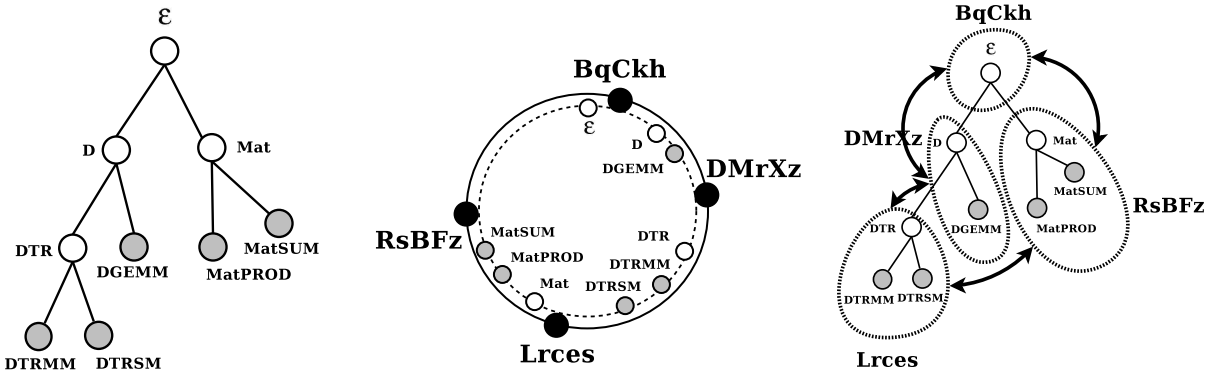
Another view of this *locality preserving Chord-like placement* of nodes on peers is given by Figure 4.1(c). Peers (pseudo-ellipses in dashed lines) are connected in a ring (bold arrowed lines), each peer running the set of nodes inside its ellipse. For the sake of clarity, we give another examples with BLAS routines' names, in Figure 4.2.



a. Binary PGCP Tree b. Mapping result (ring view) c. Mapping result (tree view)

Figure 4.1: Example of mapping - binary identifiers.

We now expose the details of the protocol.



a. *Latin BLAS* PGCP Tree b. Mapping result (ring view) c. Mapping result (tree view)

Figure 4.2: Example of mapping - BLAS routines.

4.2 Protocol

We consider a set of digits $\mathcal{A}$ and a circular identifier space $\mathcal{I}$ of all distinct possible identifiers i such that i is a finite sequence of digits of $\mathcal{A}$. The protocol is made of two distinct parts.

A Ring Overlay for Peers. The first part of the protocol maintains the physical network. The overlay built with peers of the physical network is a bidirectional ring over the peers, dynamically constructed as peers join and leave the network. Denote $\mathcal{P} \subseteq \mathcal{I}$ the set of peer identifiers in the ring at a given time. Peers are ordered in a bidirectional ring. Each peer $P \in \mathcal{P}$ has the knowledge of its immediate predecessor $pred_P$ and immediate successor $succ_P$ *i.e.,* peers whose identifier is the highest lower than P and the lowest higher than P, respectively. Let $P_{max} \in \mathcal{P}$ and $P_{min} \in \mathcal{P}$ be the two peers whose ids are the highest and lowest in the ring, respectively. As we will see, even if the process of inserting or removing a peer aims at maintaining the physical network, the routing of the insertion request itself is mainly achieved within the logical connections of the tree.

A Logical PGCP Tree to Index Information. The other part maintains a Greatest Common Prefix Tree over services' keys as services are declared by their server providers. Denote $\mathcal{N} \subseteq \mathcal{I}$ the set of node identifiers currently in the tree. The registration of the service $s = (k, v)$ leads to the creation of one node if $\neg(\exists n \in \mathcal{N} \ : \ n = k)$. The protocol maps the tree onto the peers as it is growing. The mapping scheme ensures that the peer P chosen to run a given node n always satisfies the condition that P is the *successor peer* of n, *i.e.,* whose identifier is the lowest higher than n. We denote $succ(n)$ the peer running n. Note that the identifier space is a *circle*. As a consequence, $\forall n \in \mathcal{N}$ such that $n > P_{max}$, $succ(n) = P_{min}$. Each node n maintains a father f_n, a set of children C_n and the set of all values δ_n associated with its label, the key l_n. Finally, we assume that the number of nodes is greater than the number of peers and that each peer runs at least one node (we explain how we maintain such a property in the following).

We now have a clearer idea of what we find on one peer, which is illustrated on Figure 4.3: The whole ellipse represents a processor which takes part in the system. Inside, we find at least one peer (P) connected on the ring to its successor and predecessor. Each peer runs a set of processes being nodes in the tree (here n_1, n_2 and n_3), each one being connected to its respective parent and children.

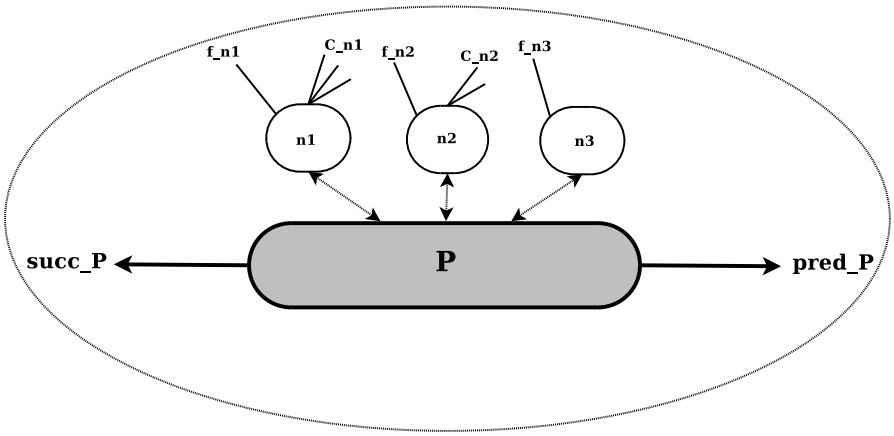


Figure 4.3: Internal architecture of a processor taking part of the system.

As before, we assume two basic functions.

- PREFIXES(k) returns the set of ids properly prefixing k. For instance, PREFIXES(10101) returns $\{\epsilon, 1, 10, 101, 1010\}$
- GCP(k_1, k_2) returns the greatest common prefix shared by k_1 and k_2. For instance, GCP$(101, 100) = 10$.

4.2.1 Peer insertion

When a peer Q joins the system, the routing of the join request sent by Q is first received by the contact peer and is then handled by the nodes, starting from one node chosen (the closest to the identifier of the joining peer) by the contact peer among the set of nodes it runs. The routing process is then similar to those of the previous chapter. The request reaches a node run on a peer close to the final destination of Q. Then the effective insertion is performed. This protocol is detailed by algorithms 3 and 4. The sought peer is the one with the highest identifier lower than Q. To reach this peer, we first route the request to the node with the highest id lower than Q.

In details, the path of a PEERJOIN request is made of three steps. During a first step (Lines 2.02 to 2.09), the request is marked 0, moves upward and eventually reaches a node that is either a prefix of Q or the root of the tree, what updates the state of the request to 1. During a second step, the request is thus marked 1 (lines 2.11 to 2.13) and moves downward until reaching the target node t whose label is the highest lower than Q (Line 2.11 repeatedly finds the greatest child lower than Q). t then sends the request to the peer on which it runs (and successor on the identifier circle, and the request is delegated to the peer layer (Line 2.15). This communication between layers is illustrated in Figure 4.3 by double-arrowed lines between the nodes and the peer.

The final step consists in deciding whether Q shall be a predecessor of T, or a predecessor of $succ_T$ (what is tested Line 4.02). Only these two cases are possible since $pred_T < Q \leq succ_T$. $pred_T < Q$ comes from the facts $pred_T \leq t$ and $t \leq Q$. This Q can not be a predecessor of $pred_T$. Now, by contradiction, let's assume $Q > succ_T$. Since ν_T, the set of nodes run on Q is assumed not empty, $\exists n \in \nu_T$ such that $n > t$, which means that the first part of the algorithm (finding the target

node) did not give the proper t. This is impossible as we easily showed that the algorithm did. A contradiction.

Once decided whether $succ_Q$ is T or $succ_T$, it remains to effectively insert Q and dispatch the set of services ν_{succ_Q} until now stored at $succ_Q$, among Q and $succ_Q$, according to their keys, as detailed lines 4.05-4.09. The YOURINFORMATION message contains the information required for Q to run *i.e.,* $(pred, succ, nodes)$. The UPDATESUCCESSOR message informs $pred_Q$ that its successor has changed (from T) to Q (see the receipt of these messages in Lines 5.01-5.04 and Lines 6.01-6.02.

Algorithm 3 Peer insertion, on node n

```
1.07  Variables:  l_n, label of n
                  f_n, identifier of the father of n
                  C_n, set of pairs (identifier, label) of children of n

2.01  upon receipt of <PeerJoin, (Q, s)> do
2.02      if s = 0 then
2.03          if Q ∉ Prefixes (l_n) then
2.04              if (f_n = ⊥) then
2.05                  send <PeerJoin, Q, 1> to n
2.06              else
2.07                  send <PeerJoin, Q, 0> to f_n
2.08          else
2.09              send <PeerJoin, Q, 1> to n
2.10      else
2.11          q =Max ({c ∈ C_n : c ≤ Q})
2.12          if (q ≠ ⊥) then
2.13              send <PeerJoin, Q, 1> to q
2.14          else
2.15              send_to_host<NewPredecessor, Q>
```

Peer removal. When a peer wants to leave the network, the nodes that were running on it are just handled by its successor, and the successor's and predecessor's variables are easily updated, as the leaving peer gives them the information needed. In the case a peer leaves without notice, some periodic scanning mechanisms from the successor and predecessor similar to those used in Chord [135] can easily detect it. As done in Chord, maintaining links to their k successors on the ring allow to handle multiple peers leaving at the same time.

4.2.2 Service registration

This part is similar to the one described in Section 3.3, except that now we do not rely on any subsystem able to return peers. In other words, our overlay maps nodes *itself.*

To declare the availability of a service $s = (k, v)$ identified by k, a peer (or server) sends a SERVICEINSERTION request to a random node of the tree. The protocol routes the request to the node with the identifier closest to k. If $\nexists n \in \mathcal{N}$ such that $l_n = k$, such a node is created, inserted in the tree and run on a peer. In any case, v is eventually added to the set of data δ_n of the node n with $l_n = k$. This process is detailed in Algorithm 5, Page 75. Similarly as in the previous chapter, on receipt of the request, the node p proceeds according one of the four following cases (as

Algorithm 4 Peer Insertion, on peer P

```
3.01  Variables:  succ_P, successor of P
                  pred_P, predecessor of P
                  ν_P, set of nodes running on P

4.01  upon receipt of <NewPredecessor, Q> do
4.02      if Q > P then
4.03          send <NewPredecessor, Q> to succ_P
4.04      else
4.05          ν_Q = {n ∈ ν_p : n ≤ Q}
4.06          ν_P = {n ∈ ν_p : n > Q}
4.07          send <YourInformation, (pred_P, P, ν_Q)> to Q
4.08          send <UpdateSuccessor, Q> to pred_P
4.09          pred_P := Q

5.01  upon receipt of <YourInformation, (pred, succ, nodes)> do
5.02      pred_P := pred
5.03      succ_P := succ
5.04      ν_p := nodes

6.01  upon receipt of <UpdateSuccessor, Q> do
6.02      succ_P := Q
```

the maintenance of the tree has already been detailed in the previous chapter, we briefly recall the registration algorithm and then emphasize the mapping part of the protocol):

- If $l_n = k$ (Line 8.02), n is the sought node. v is added to δ_n (no node needs to be created.)
- If $l_n \in$ Prefixes (k) (lines 8.03 to 8.08), the sought node is in the subtree of n. If $\exists q \in C_n$ such that the label of q shares a longer prefix with k than l_p does, the sought node is in the subtree rooted at q and the request is forwarded to q. Otherwise, the sought node does not exist and is created as a child of p. To find a host for the new node, the complete information for this node $(label, values, parent, set\_of\_children)$ is sent to p itself using the SearchingHost message. This part of this protocol is detailed later.
- If $k \in$ Prefixes (l_n) (lines 8.09 to 8.20), the sought node is upward. If k is also a prefix of l_{f_n}, then the request is forwarded to f_n. Otherwise, the sought node does not exist and is created between n and f_n (the new node become the root of the tree if $f_n = \perp$). Details of receipt of AddChild and RemoveChild messages are not detailed because trivial, following the details of the previous chapter.
- Finally, if none of the previous cases is satisfied, the algorithm behaves similarly than for the previous case. If l_{f_n} shares the same prefix with k as l_n, the request is again forwarded to l_{f_p}. Otherwise, the sought node does not exist. n and the node labeled k are siblings, but their common parent also does not exist. Two nodes are created, one to store k and one to preserve the prefix patterns inside the tree, common parent of n and k, labeled by GCP(l_n, k).

New nodes are created at different lines of Algorithm 5. When created, a new node, say nn, labelled l_{nn}, must find the peer on which it will run. As mentioned earlier, the

$$\text{SEARCHINGHOST}(key, values, parent, parent\_label, set\_of\_children)$$

message initiates the search for a peer to host nn. This part is detailed by lines 9.01 to 9.09. The algorithm is designed such as the first recipient of the SEARCHINGHOST message always prefixes l_{nn}. Sometimes the search starts at n, sometimes at its parent f_n. For instance, on Line 8.28, the new node is labeled GCP(l_n, k) and is lower than l_n but greater than l_{f_n}. Then, we are sure that the peer we want already runs some nodes which is in the subtree of f_n (the process to find the successor of a peer is very similar to the one to find the successor peer of a node). Once the request is initiated close enough to the root, it suffices to move the request downward until reaching the highest node lower than l_{nn}. With Line 9.09, the right peer receives the complete information required to host nn and starts a new process for this node. Once the new process is launched, the lower layer sends back an acknowledgement to the upper layer with the identifier of the new process (the address of the process — typically, the address of the peer and the port of the new process). The node, on receipt of the acknowledgement from its lower layer, sends an acknowledgement to the node which initiates the search for a host by synchronously following the reverse path to the initiator. The initial nodes can then finish the insertion by informing all nodes pertained by the insertion, *i.e.,* sends the ADDCHILD and REMOVECHILD messages and update its own variables.

4.2.3 Load Balancing

Each peer runs a set of nodes. As detailed before, the routing follows a top-down traversal. Therefore, the upper a node is, the more times it will be visited by some request. Moreover, due to the sudden popularity of some service, the nodes storing the corresponding keys, independently from their depth in the tree, may become overloaded. The heuristic we present now deals with this issue by maximizing the aggregated throughput of two consecutive peers, *i.e.,* the number of requests these two heterogeneous peers will be able to process. This is achieved by periodically redistributing the nodes on the peers, based on recent history.

For the sake of clarity, we consider a discrete time and choose one particular peer S to illustrate the process. The load balancing process is triggered on S at the end of each unit of time. Let $P = pred_S$ be the predecessor of S. Refer to Figure 4.4(a). C_S and C_P refer to their respective capacities, *i.e.,* the number of requests they are respectively able to process during one unit of time. Note that the peers capacity does not change over time. At the end of a time unit, some peers send the number of requests they received during this time unit, for each node it runs, to its successor. Here, S has information on the loads of the nodes run by P, and obviously its own load information. Assume that, during last time unit τ, the set of nodes run by S and P were respectively ν_S^τ and ν_P^τ and that each $n \in \nu_S^\tau \cup \nu_P^\tau$ has received a **different** number of requests l_n. Then, the load, denoted L_S^τ of S during the period τ was the sum of the loads of the nodes it runs, *i.e.,*

$$L_S^\tau = \sum_{n \in \nu_S^\tau} l_n. \tag{4.1}$$

We easily see that, during this time unit τ, the number of satisfied requests (or *throughput* T), *i.e.,* requests that were effectively processed until reaching its destination (or finding the service sought does not exist) is:

$$T_{S,P}^\tau = min(L_S^\tau, C_S) + min(L_P^\tau, C_P). \tag{4.2}$$

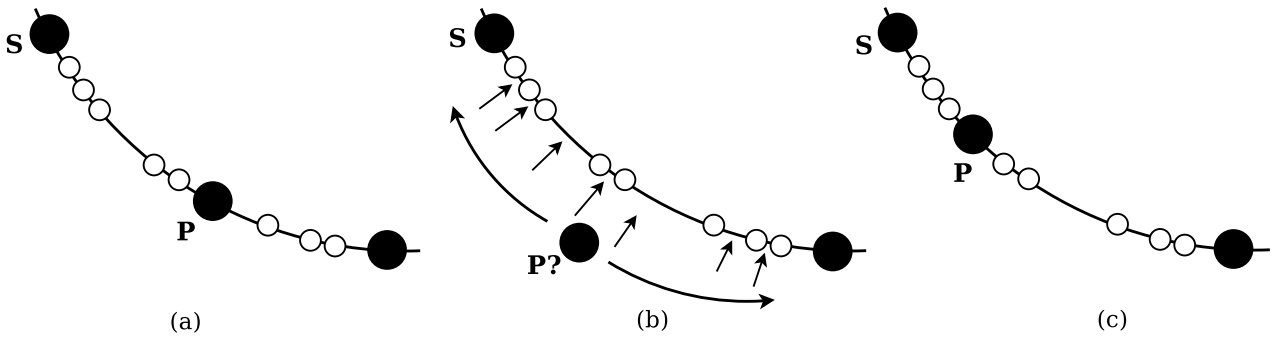


Figure 4.4: One local load balancing step.

Starting from this knowledge, *i.e.,* the load of every nodes $n \in \nu_S^\tau \cup \nu_P^\tau$, we want to maximize the throughput of the next unit of time $\tau + 1$. To do so, we need to find the new distribution ($\nu_S^{\tau+1}$, $\nu_P^{\tau+1}$) that maximizes the throughput *i.e.,* such that

$$T_{S,P}^{\tau+1} = min(\sum_{n \in \nu_S^{\tau+1}} l_n, C_S) + min(\sum_{m \in \nu_P^{\tau+1}} l_m, C_P) \tag{4.3}$$

is maximum, assuming the load distribution will be similar in times $\tau + 1$ and τ. The number of possible distributions of nodes on peers is bounded by the fact that nodes identifiers can not be changed, in order to ensure the routing consistency and a way to retrieve services. The only parameter that we can change is the identifiers of peers. Then, as illustrated on Figure 4.4, finding the best distribution is equivalent to find the *best position* of P moving along the ring, as illustrated by arrows on Figure 4.4(b). The number of candidate positions for P is $|\nu_S \cup \nu_P| - 1$. Thus, the time and extra space complexity of the redistribution algorithm is clearly in $O(|\nu_S^\tau \cup \nu_P^\tau|)$. In other words, even if periodically performed, the MLT heuristic has, locally, a constant communication cost and a time complexity linear in the number of nodes between the two local peers. An example of the result of this process is given by Figure 4.4(c), where, according to the previous metric, the best distribution is three (weighted) nodes on S, and 5 (weighted) nodes on P. This heuristic is henceforth referred to as *MLT* (*Max Local Throughput*).

4.3 Simulation

To validate our approach, we developed a simulator of this architecture, into which we integrated two load balancing heuristics: *MLT* and an adaptation of a recent load balancing algorithm initially designed for DHTs known as the *k-choices* algorithm [95] and, to our knowledge, the most related existing heuristic. We denote *KC* this adaptation. More precisely, when used, *KC* is run each time a peer joins the system. Because some regions of the ring are more densely populated than others, *KC* finds, among k potential locations for the new peer, the one that leads to the best local load balance. Please refer to [95] for more details.

We used a discrete time in the simulations. One simulation was made of a fixed number of time units. Each simulation were repeated 30, 50 or 100 times, to have some relevant results. Recall that the *capacity* of a peer refers to the maximum number of requests processed by it during one time unit. All requests received on a peer after it has reached this number are ignored and considered as

unsatisfied. The ratio between the most and the least powerful peer is 4. A request is said to be *satisfied* if it reaches its final destination. The number of peers is approximately 100, and the number of nodes around 1000. We set *KC* with $k = 4$ (which we believe is relevant, since, as established by Azar, Broder, Karlin and Upfal [24], beyond $k = 2$, the gain on the load balance is just a constant factor). As in the previous chapter, the prefix trees are built with identifiers commonly encountered in a grid computing context such as names of linear algebra routines.

We first estimated the global throughput of the system when using *MLT*, *KC*, and no explicit load balancing at all. Each time unit is composed of several steps. (1) In experiments where *MLT* is enabled, a fixed fraction of the peers executes the *MLT* load balancing. (2) A fixed fraction of peers join the system (applying the *KC* algorithm in experiments where it is enabled, or simply the basic protocol detailed in Section 4.2, otherwise.) (3) A fixed fraction of peers leaves the system. (4) A fixed fraction of new services are added in the tree (possibly resulting in the creation of new nodes). (5) Discovery requests are sent to the tree (and results on the number of satisfied discovery requests are collected).

During first experiments, services requested were randomly picked among the set of available services. Figure 4.5 gives the percentage of satisfied requests using *MLT*, *KC* and no load balancing, for 50 time units. The first 10 units correspond to the period where the prefix tree is growing. After, it remains the same. The obtained curves show that using heuristics, and more particularly *MLT*, leads to a non negligible gain. Figure 4.6 shows the results of the same experiment, but with a very high number of requests, in order to stress the system. We observe similar results, even if the satisfaction percentage is obviously globally lower.

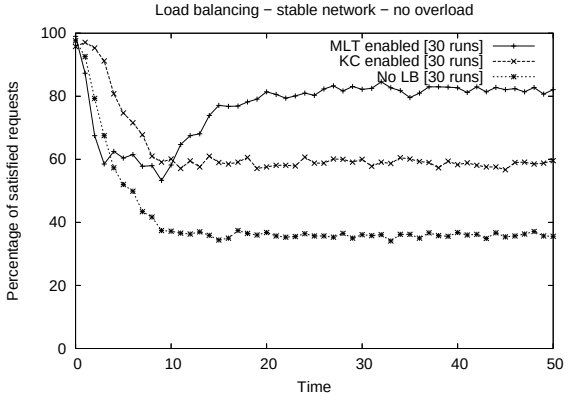


Figure 4.5: Stable network, low load.

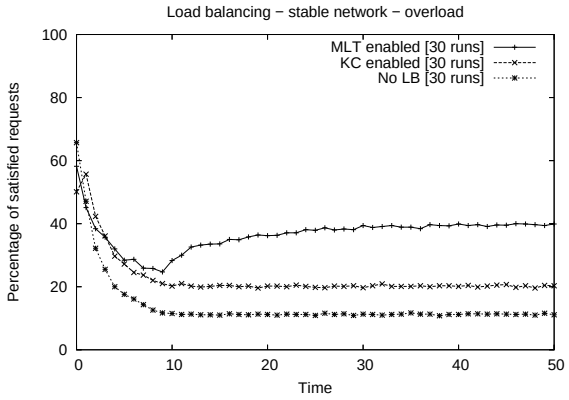


Figure 4.6: Stable network, high load.

Until now, experiments were conducted in a relatively stable network. The number of peers joining and leaving the system was intentionally low. The efficiency of the *KC* algorithm relies on the dynamic nature of the system since load balancing is done each time a peer joins the system. Now, 10% of the nodes are replaced at each time unit. This is why we repeated these experiments with an increased fraction of peers joining and leaving the network. Figures 4.7 and 4.8 give the results of the same experiments than before but conducted over a dynamic platform. We see that *KC* performs a bit better than previously, and gives results similar to *MLT*.

We conducted these experiments for different loads. The results are summarized in Table 4.1. The percentages in the left column express the ratio between the number of requests and the aggregated

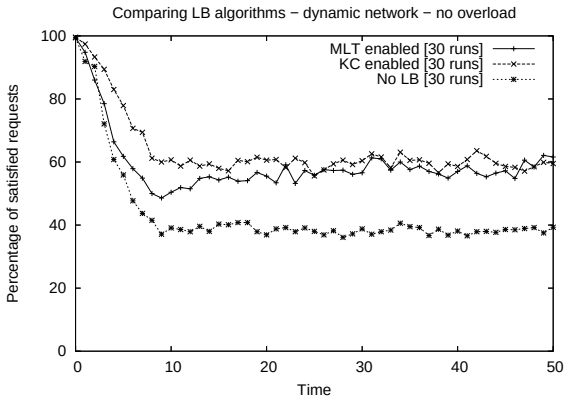


Figure 4.7: Dynamic network, low load.

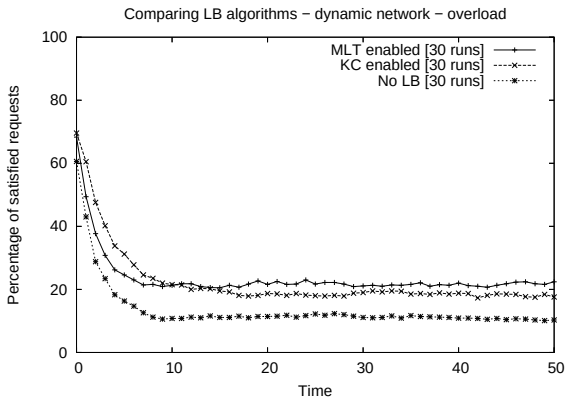


Figure 4.8: Dynamic network, high load.

Load	Stable network		Dynamic network	
	MLT	KC	MLT	KC
5%	39,62%	38,58%	18.25%	32,47%
10%	103,41%	58,95%	46,16%	51,00%
16%	147,07%	64,97%	65,90%	59,11%
24%	165,25%	59,27%	71,26%	60,01%
40%	206,90%	68,16%	97,71%	67,18%
80%	230,51%	76,99%	90,59%	71,93%

Table 4.1: Summary of gains of *KC* and *MLT* heuristics.

capacity of all peers in the system. The table gives the gain on the number of satisfied requests of each heuristic compared to the architecture with no load balancing. As we can see reading the table, the gain can be really important, and the *MLT gain* is globally higher than the *KC gain*.

Our last series of simulations, whose result is illustrated by Figure 4.9, consisted in creating hot spots in the tree, by temporarily launching many discovery requests on some keys stored in the same region of the tree, *i.e.,* on lexicographically closed keys, in bursts. The experiment is divided in time as follows: during the first 40 time units, services are again randomly picked. Then, between 40 and 80, a hot spot is created on the particular S3L library. Most of S3L routines are named by a string beginning by "S3L". We thus overloaded the subtree containing the keys prefixed by "S3L". The network was previously balanced for random requests, and as a consequence, the number of satisfied requests suddenly falls. However, the *MLT*-enabled architecture adapts to the situation and increases the satisfaction ratio to a reasonable point, by moving more peers in the region of the "S3L*" nodes. A second change arises at time 80, when simulating the arrival of many requests on the ScaLAPACK library whose functions begin with "P". The system reacts again and provide an improved throughput again. The random way to pick services is chosen for the 40 last time units, leading to a behavior similar to the one of the beginning. As less hotspots are created, the throughput quickly improves to reach the performance level of the beginning of the simulation.

Functionality	P-Grid	PHT	DLPT
Tree Routing	$O(log\|\Pi\|)$	$O(D \log P)$	$O(D)$
Local State	$O(log\|\Pi\|)$	$\frac{\|N\|}{\|P\|}\|A\|$	$\frac{\|N\|}{\|P\|}\|A\|$

Table 4.2: Complexities of close trie-structured approaches.

Avoiding Physical Communications by *Lexicographic Clustering*. Finally, as previously said, the mapping scheme is better in several ways than a random DHT-based mapping, since a random mapping results in breaking the locality of keys. Connected nodes in the tree are randomly dispatched in random locations of the physical network. With our mapping scheme, the set of nodes stored on one peer are highly connected. This fact brings about a reduction of the communications between peers, since a high amount of routing steps in the tree involves two nodes running on the same peer. Figure 4.10 gives, for each time unit, and following the same experimental scheme as in simulation with hot-spots, the average number of hops in the tree required to reach their final destination. The total number of hops is provided (between 8 and 10 for each request). We see that our self-contained mapping featured with *MLT* significantly reduces the amount of communications within the physical network (from approximately 9 to approximately 3), while the random DHT-based mapping reduces it only by approximately 1.

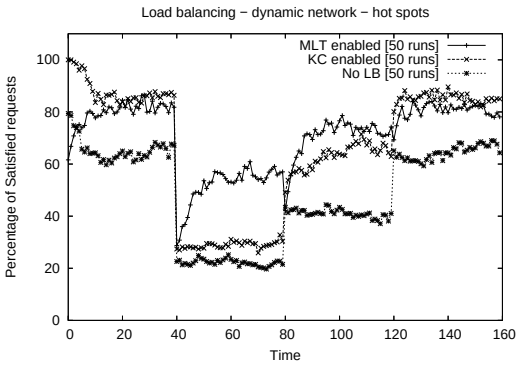


Figure 4.9: Dynamic network with hot spots.

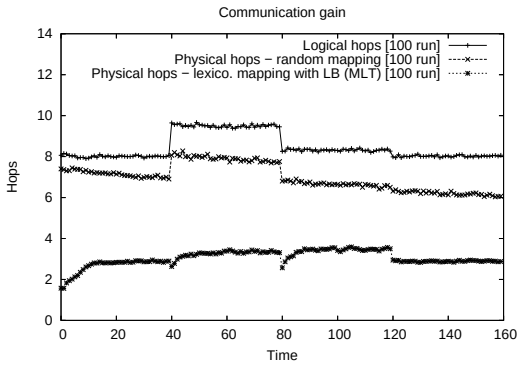


Figure 4.10: Reduction of the communication by the *lexicographic* mapping.

4.4 Comparison to Related Work

A fair comparison of the complexities of our architecture with PHT and P-Grid can now be achieved (see Table 4.2.) The complexities of our self-contained approach and the two previously mentioned approaches are quite similar. $|\Pi|$ refers to the number of partitions of the key-space, D to the maximal length of the identifiers, A to the set of digits used, N to the set of nodes of the tree and P to the set of peers.

In regard of this chapter, One of our contribution over PHT and P-Grid is in the load balancing process. The PHT load balancing assumes the peers homogeneous. (They have a fixed capacity). It relies on a global threshold on the number of keys each node maintains. P-Grid relies on a set

of algorithms periodically checking the load balance. In these two approaches, the heterogeneous capacities of the peers and the popularity of services/resources are ignored. We believe that the number of keys maintained on a node does not accurately reflects its load, since it depends on what users request. Moreover, the capacities of the peers can not be supposed homogeneous in a grid computing context. For these reasons and because our architecture maintains a *ring* over the peers, we studied the works on the load balancing issue within DHTs.

Among the numerous papers about the load-balancing in DHTs, Karger and Ruhl [90] and similar approaches from which you can find the details in Chapter 2, propose some local heuristics based on item balancing, but assume homogeneous peers. Even if Godfrey *et al.* [71] tackle the **weighted** problem, they use a set of elected nodes gathering the load information and redistributing items with partial knowledge of loads and capacities. The drawback of this approach is clearly its *semi-centralized* fashion, which assumes some peers to be reliable, at least in the period during which they compute the redistribution of items on peers. A load balancing strategy for Chord proposed in [35] uses multiple hash function to select a number of candidate peers, based on the well-known *power of two choices* paradigm. Ledlie and Seltzer [95] proposed the similar *k-choices* approach, but assuming heterogeneity of both peers and items. We showed by simulation that our heuristic, based on the local maximization of the throughput can outperform this last algorithm.

4.5 Conclusion

In this chapter, we focused on improving two aspects of DLPT: mapping and load balancing. The first contribution is a complete protocol for a self-contained version of this architecture and the avoidance of the use of an underlying DHT. Our overlay only needs tree connections and a degree 2 to build a ring over the peers. We provided the detailed message passing algorithms along with all information to prove their correctness. This mapping requires only an extra-cost of two connections by peer, in addition to the tree connections themselves. Such a lexicographic mapping allow to *cluster* many tree links inside one peer, thus drastically reducing the communication cost.

The second contribution is a novel heuristic for the load balancing inside this architecture and the adaptation to our case of recent techniques initially designed for the same purpose within DHTs. Different simulations show the important gain obtained by using these heuristics. We have finally proposed a comparison of close approaches, in terms of cost and load balancing.

Algorithm 5 Service insertion, on node n

7.01 **Variables:** l_n, the label of n, δ_n, set of values stored on n
f_n, identifier of the parent of n, l_{f_n}, label of the parent of n
C_n, set of pairs $(identifier, label)$ of children of n

8.01 **upon** receipt **of** <ServiceInsertion, $s = (k, v)$> **do**
8.02 **if** $k = l_n$ **then** $\delta_n := \delta_n \cup \{v\}$
8.03 **elseif** $l_n \in$ Prefixes (k) **then**
8.04 **if** $\exists q \in C_n$: $|\text{GCP}(k, l_q)| > |\text{GCP}(k, l_n)|$ **then send** <ServiceInsertion, s> **to** q
8.05 **else**
8.06 **send** <SearchingHost, $(k, \{v\}, n, l_n, \emptyset)$> **to** n
8.07 **receive** <SearchingHostDone, $newID$> **from** n
8.08 $C_n := C_n \cup \{(newID, k)\}$
8.09 **elseif** $k \in$ Prefixes (l_n) **then**
8.10 **if** $(f_n = \perp)$ **then**
8.11 **send** <SearchingHost, $(k, \{v\}, \perp, \perp, \{(n, l_n)\})$> **to** n
8.12 **receive** <SearchingHostDone, $newID$> **from** n; $f_n := newID$; $l_{f_n} := k$
8.13 **else**
8.14 **if** $k \in$ Prefixes(l_{f_n}) **then send** <ServiceInsertion, s> **to** f_n
8.15 **else**
8.16 **send** <SearchingHost, $(k, \{v\}, f_n, l_{f_n}, \{(n, l_n)\})$> **to** f_n
8.17 **receive** <SearchingHostDone, $newID$> **from** f_n
8.18 **send** <RemoveChild, (n, l_n)> **to** f_n
8.19 **send** <AddChild, $(newID, k)$> **to** f_n
8.20 $f_n := newID$; $l_{f_n} := k$
8.21 **else**
8.22 **if** $(f_n \neq \perp) \wedge (|\text{GCP}(k, l_n)| = |\text{GCP}(k, l_{f_n})|)$ **then send** <ServiceInsertion, s> **to** f_n
8.23 **else**
8.24 **if** $(f_n = \perp)$ **then**
8.25 **send** <SearchingHost, $(\text{GCP}(l_n, k), \emptyset, \perp, \perp, \{(n, l_n)\})$> **to** n
8.26 **receive** <SearchingHostDone, $newID1$> **from** n
8.27 **else**
8.28 **send** <SearchingHost, $(\text{GCP}(l_n, k), \emptyset, f_n, l_{f_n}, \{(n, l_n)\})$> **to** f_n
8.29 **receive** <SearchingHostDone, $newID1$> **from** f_n
8.30 **send** <RemoveChild, (n, l_n)> **to** f_n
8.31 **send** <AddChild, $(newID1, \text{GCP}(l_n, k))$> **to** f_n
8.32 **send** <SearchingHost, $(k, \{v\}, newID1, \text{GCP}(l_n, k), \emptyset)$> **to** f_n
8.33 **receive** <SearchingHostDone, $newID2$> **from** f_n
8.34 **send** <AddChild, $(newID2, k)$> **to** $newID1$
8.35 $f_n := newID1$; $l_{f_n} := \text{GCP}(l_n, k)$

9.01 **upon** receipt **of** <SearchingHost, (l, δ, f, l_f, C)> **from** r **do**
9.02 $q =$Max$\{c \in C_n$: $c \leq l\}$
9.03 **if** $(q \neq \perp)$ **then**
9.04 **send** <SearchingHost, (l, δ, f, l_f, C)> **to** q
9.05 **receive** <SearchingHostDone, $newID$> **from** q
9.06 **else**
9.07 **send_to_host**<Host, $(l, \delta, f, C, \delta)$>
9.08 **receive_from_host**<HostDone, $newID$>
9.09 **send** <SearchingHostDone, $newID$> **to** r

Chapter 5

Fault-Tolerance and Self-Stabilization

In this chapter[1], we present the *best-effort* alternatives we developed to allow to recover a consistent service discovery service after a number of crashes higher than the replication can handle. The consequence of a too high number of crashes is the impossibility to route requests, due to an inconsistent topology. Even if the information on services have been possibly partly lost, here, we want to provide users with a discovery systems available and able to process requests on the information remaining, even if the system is continuously undergoing an important number of failures/crashes, without the need of a complete reset of the system. Three approaches are presented:

1. **Reconnecting and Reordering Valid Disconnected Subtrees.** Our first goal is to be able to recover a valid tree after physical node crashes. Our first attempt was to develop an algorithm *repairing* a tree after an arbitrary set of crashes leading to the loss of nodes and the split of the tree into a forest. However, the remaining disconnected subtrees are assumed valid. Our first algorithm, written as a message-passing protocol, is detailed in Section 5.1.

2. **Snap-Stabilizing Rooted Connected Prefix Tree.** Our first approach is not *self-stabilizing* in the sense that the subtrees to be reconnected are assumed to be valid. In other words, if subtrees about to be reconnected are not valid, or if, for some reason, some variables, *e.g.,* neighbors' references, on the nodes are not correct, the protocol is unable to systematically rebuild a valid tree. We have thus designed a *snap-stabilizing* protocol to maintain a PGCP tree. Recall that *snap*-stabilizing means that the protocol always behaves according to its specification. Only a constant number of initializations of the protocol is required to have a consistent tree again. This protocol, written in the theoretical state model, is detailed in Section 5.2.

3. **Self-Stabilizing Message Passing Prefix Tree Starting from an Arbitrary Configuration.** Although the previous protocol is optimal in stabilization time, it suffers of several drawbacks in terms of applications on a real platforms. First, the previous protocol being written in the restricted theoretical state model, we know that the correctness of the algorithm can not be proved straightforwardly in a message-passing environment [110, 50]. Moreover, this protocol assumes that the topology is always a rooted connected tree, what can be ensured only if a first mechanism *glued* the subtrees together. To address these drawbacks, we developed a third protocol, that only need a message passing environment and that we proved to

[1]The work presented in this chapter has been published in international conferences [CDFPT06,CDPT07,CDPT08] and national conferences [CFPT06,Ted08].

be self-stabilizing. In this last work, detailed in Section 5.3, we studied the pragmatic impact of this self-stabilizing protocol on the fault-tolerance of such architectures.

5.1 Reconnecting and reordering valid subtrees

In this section, we present a message passing protocol reconnecting and reordering distributed PGCP subtrees after the crashes on some peers, and, as a consequence, the loss of nodes in the tree and the split of the logical indexing system in a forest.

5.1.1 Preliminaries

The definition of a *valid* tree is given by Definition 1, Page 45.

The *distributed system* considered in this section consists of a set of asynchronous physical nodes (processors, or *peers*) organized in a *Distributed Hash Tables (DHT)*. The mapping scheme of Chapter 4 is not taken into account. Each peer maintains one or more nodes of a logical PGCP Tree. Note that, as in Chapter 3, a DHT is used, but can be replaced by any system, distributed or not, allowing the retrieval of a peer's reference. In the same vein, we also consider that the potentially existing fault-tolerance mechanisms provided by this layer are not used, since our algorithm provides the fault-tolerance at the tree layer. The DHT is just a mean to obtain peers' references, as in Chapter 3.

Nodes of the logical layer (trees) communicate by *message passing*. We assume two sending functions. The former, simply referred to as **send**, is used by any node to send a message to another node asynchronously, *i.e.,* without waiting any acknowledgement. The latter, called **send-sync**, waits for an acknowledgement for each message sent. We assume that each physical node may crash. When a physical node crashes, one or more logical nodes are lost.

As discussed earlier and addressed by several theoretical papers, like the paper of Shaker and Reeves [129], we need a weakly-connected bootstrap mechanism. In other words, disconnected subtrees have no knowledge of each others. The only way to ensure to have a connected logical network again is to rely on a subsystem able to gather all remaining nodes in the system. Optimizing the average number of calls made to this system is here out of topic. That's why, for the bootstrap problem, we just use the DHT to collect peers (and nodes running on peers), broadcasting the entire DHT if required.

5.1.2 Protocol

In this section, we give a detailed explanation of how the protocol works. We divide the algorithm code in two parts. The former shows the first phase developed with our technique during which a unique tree is recovered without considering any lexicographic property. During the second phase, the trie is reorganized to eventually form a distributed greatest common prefix tree.

Tree Reconnection

After a node p detects the loss of its parent (f_p), it searches for a new parent to link on. Making a traversal of the DHT, Node p collects, in the variable P, the addresses of all remaining peer. Starting from the addresses in P, p collects the set of logical nodes N stored by the peers in P. Next, using a PIF (*Propagation of Information with Feedback*) Protocol [45, 128], a wave algorithm gathering

information from a whole tree starting from the root, p computes T, the set of logical nodes in its own subtree.

This first step of the reconnection protocol, detailed in Algorithm 6, Page 6, ends when p chooses a temporary parent (tf_p) in the subset $N \setminus L$. When a node q is linked to a node p, then p considers q as a temporary child—stored in T_p. Note that nodes in the variable T_p are taken into account to compute L using a PIF in the subtree of p. If $N \setminus L = \emptyset$, *i.e.,* there is no node which p may connect to), then p is considered as the "real" root of the tree.

The above technique suffers of a drawback. Several nodes without parent may make a choice which could become a "bad" choice. In particular, they can choose as a temporary parent a node belonging to the subtree of another node being in the same situation. By doing this in parallel, cycles may appear. Our strategy is to detect and to break *a posteriori* such cycles following a simple mechanism. (Again, our purpose is not performance but functionality.)

After the choice of its temporary parent tf, a node p sends a message HELLO with its ID (p) to tf. In the next step, tf transmits the message to its own parent, and so on. Step by step, one of the two following situations eventually arises:

1. The *real* root of the tree receives the message HELLO. In this case, the root notifies p that it is not involved in a cycle.

2. The message is received by a *false* root, *i.e.,* a node having also lost its own parent. The false root propagates the message to its temporary parent.

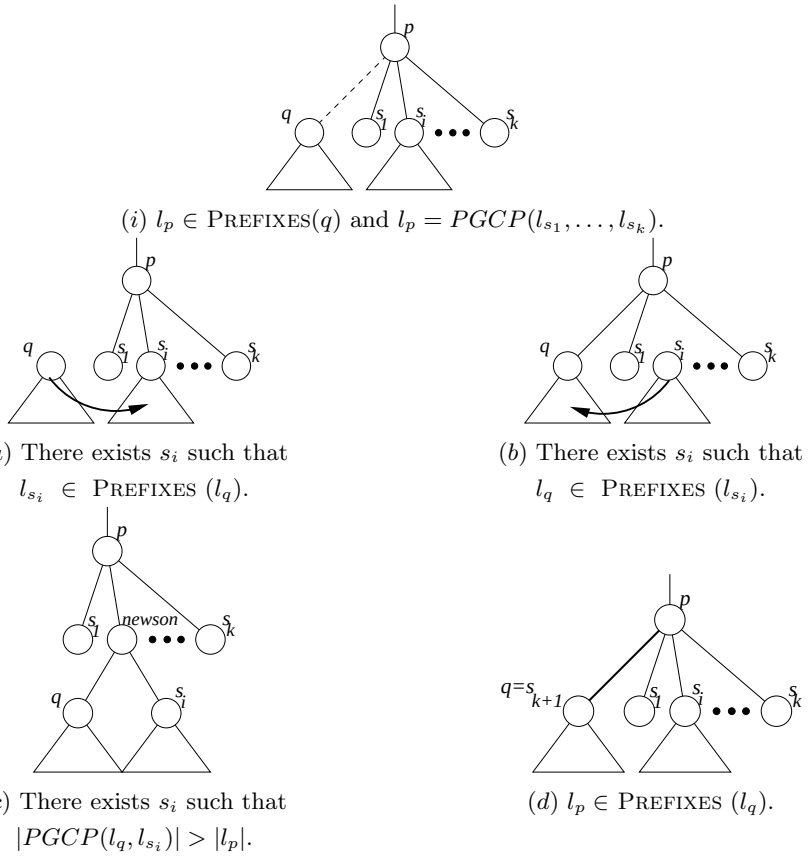
Note that, in the above latter case, due to asynchrony of the network, it is possible that the false root receives the message HELLO sent by p before it executed its own reconnection phase. In that case, a false root is still without a temporary parent. The message HELLO is then delayed until the false root chooses its own temporary parent.

Therefore, the message HELLO sent by p keeps circulating among its ancestors, carrying the list of false roots' identifiers which were met during its traversal. Upon receipt of a message HELLO, if the first item of the list carried by the message is equal to the identifier of the receiver, then a cycle is detected. In that case, a leader election is computed among the IDs of the list, *e.g.,* by choosing the smallest ID. The leader becomes the root of the subtree, breaks its link to its parent, and restarts the reconnection phase (the other *false* roots involved in the cycle remain connected to the subtree rooted by the leader). Note that a cycle may be created again. However, in the worst case, each time the reconnection phase is launched, at least one subtree becomes part of the subtree of one false root. In other words, the number of cycles is periodically divided by at least 2. Therefore, the system eventually contains one (rooted) tree only.

Tree Reorganization

The tree reorganization, detailed in Algorithm 7, Page 103 is initiated by the message MOVE, sent by a false root which found a temporary parent and then launches its part of the reorganization phase, which consists in finding its real parent. Each node p receiving a MOVE message from a temporary child q, *i.e.,* q is a false root of a subtree, initiates a routing mechanism closed to the original key insertion described in Chapter 3. Let us consider the following cases (the notation is again similar to previous chapters):

1. l_p, the label of p, is a prefix of l_q, the label of q–see Figure 5.1, Case (i). In that case, q (and its subtree) is placed in the subtree of p following one of the four cases shown in Figure 5.1, Cases (a) to (d).



(*i*) $l_p \in$ PREFIXES(q) and $l_p = PGCP(l_{s_1}, \ldots, l_{s_k})$.

(*a*) There exists s_i such that $l_{s_i} \in$ PREFIXES (l_q).

(*b*) There exists s_i such that $l_q \in$ PREFIXES (l_{s_i}).

(*c*) There exists s_i such that $|PGCP(l_q, l_{s_i})| > |l_p|$.

(*d*) $l_p \in$ PREFIXES (l_q).

Figure 5.1: A false root q is linked to a node p such that $l_p \in$ PREFIXES(l_q).

2. l_p is not a prefix of l_q. Then, p moves q to its parent which now takes the responsibility to place q (and its subtree).

Note that new services may be registered during the tree reconstruction. As a consequence, a new subtree may have been created at the same place where the false root initially was. Thus, our method requires to take into account that any false root being placed in the tree can meet a node having the same label. In that case, the two trees must be merged. That is the aim of the merging protocol, initiated by the sending of a message MERGE. Upon receipt of this message, a node p executes the procedure GLUING(q), which moves the children of q to p before withdrawing q from the tree (including the children of q's parent). Then, if necessary, p restarts recursively merging and placements among its children, in order to merge the whole subtrees eventually.

5.1.3 Correctness Proof

In this section, we discuss the correctness of our first fault-tolerant protocol. In order to do this, we first need to make the assumption that in the considered context, the crash frequency is low enough to make the tree fully built sometime. Even if this assumption may appear unrealistic in a peer-to-peer context, it is impossible to prove the termination of the protocol without it, since we would never have the time to rebuild if new failures constantly arise. In other words, we fairly

assume that no crash occurs after an arbitrarily long period of crashes, in order to let enough time for the tree to completely rebuild. (We will study the problem of the pragmatic efficiency of the protocol in a network that is too dynamic to converge in the last section of this chapter.)

Assumption 1. *If a node crashes at time t, then for every $t' > t$, no crash occurs.*

Lemma 1. *Under Assumption 1, the reconnection protocol (Algorithm 6) terminates, and when this occurs, the system contains one tree only.*

Proof. The validation mainly consists in showing that the protocol terminates and that the reorganization of the tree is eventually initiated (by sending a message NoCycle).

Assume by contradiction that under Assumption 1, no node eventually sends a message NoCycle. (We now refer to Algorithm 6.) So, Line 4.17 is never executed. The height of the tree being finite, this means that every Message Hello traverses only cycles and thus eventually reaches its initiator. When a message Hello is received by its initiator, the cycle is broken by the node which is elected among the false roots participating in the cycle, Lines 4.02 to 4.05. Therefore, cycles are created infinitely often. Let C be the number of created cycles. In the worst case, a cycle is made of at least two nodes. So, C is initially bounded by $F/2$, where F is the number of false roots created by the crashes. When a cycle is broken, at most one leader is elected. So, at most $C/2$ leaders are able to link another node again. In the next phase, the number of cycles is less than or equal to $C/2$. Since under Assumption 1, cycles may be created only when false roots are linked to other nodes (executing Lines 3.06 and 3.07), C never grows and is eventually equal to 0. This contradicts that cycles are created infinitely often. □

We now consider the phase of tree reorganization detailed by Algorithm 7.

Lemma 2. *Under Assumption 1 and assuming that the system contains one tree only, the reorganization protocol (Algorithm 7) terminates, and when this occurs, the tree is a PGCP tree.*

Proof. Clearly, each tree of the forest following the crash of a node is a PGCP tree (disconnected trees are assumed valid because, in this first part we only consider crashes, *i.e.,* machines leaving the system). So, it remains to show that executing Algorithm 7, the whole tree eventually satisfies the condition to be a PGCP tree.

From the algorithm, it is easy to observe that, in the absence of merging, there are only two cases to consider depending on the value of the label l_p of node p and the label l_{fs} of its false child fs :

1. l_p is a prefix of l_{fs}—Line 9.04. In that case, following the four cases described in Figure 5.1, fs is eventually placed at the right place in the subtree of p—refer to Lines 9.05 to 9.17. The resulting tree is a PGCP tree.

2. The value of p is not a prefix of fs. Again, there are two cases to consider:

   (a) Node p has no parent ($f_p = \perp$)—Lines 9.21 to 9.26. In that case, if l_{fs} is a prefix of l_p, then p (and its subtree) becomes the node to be placed by fs—Line 9.24. Otherwise, p and fs become the two children of a new root node q such that $l_q = PGCP(l_p, l_{fs})$—Line 9.26. The tree is then clearly a PGCP tree.

   (b) Node p has a parent. Then, fs is moved to the parent of p—Line 9.20. By induction of the above discussion, either fs eventually reaches a node q such that $l_q \in \text{Prefixes}(l_{fs})$ or fs eventually reaches the root of the tree. The former case is equivalent to case 1, the latter to case 2a.

If p and fs merge, then there are four cases to consider after p and fs glued together into p and p sorted its new set of children:

1. There exists a pair of children s_i, s_j of p such that l_{s_i} is a prefix of l_{s_j}. Then, s_j is moved towards s_i—Lines 10.08 to 10.11. This case is similar to the directly above case 1 (cases (a) or (b) in Figure 5.1).

2. There exists a pair of children s_i, s_j of p such that $|PGCP(l_{s_i}, l_{s_j})| > |l_p|$. Then, s_i and s_j become the two children of a new child q of p such that $l_q = PGCP(l_p, l_{fs})$—Lines 10.12 to 10.15. This case is also similar to the above case 1 (case (c) in Figure 5.1).

3. There exists a pair of children s_i, s_j of p such that $l_{s_i} = l_{s_j}$. This case is solved by initiating a recursive merging between s_i and s_j—Lines 10.05 to 10.07. This case is solved by induction on s_i and s_j.

4. There exists no pair of children s_i, s_j of p satisfying either case 1, case 2, or case 3. In that case, the subtree of p clearly satisfies the definition of a PGCP tree.

□

From Lemmas 1 and 2 follows:

Theorem 1. *Under Assumption 1, Algorithm 6 and Algorithm 7 provide a PGCP tree reconstruction after the crash of a peer.*

A First Step Towards Self-Stabilization. The protocol we have detailed in this section is a first attempt at proposing protocols for best-effort policies in prefix-tree structured peer-to-peer systems. More precisely, it allows to recover and reorganize valid subtrees to build a consistent tree again after crashes of peers. As we already mentioned, this protocol is not *self-stabilizing* in the sense that it is unable to systematically rebuild a consistent PGCP tree starting from an arbitrary configuration (with messages in links and variables on nodes in an arbitrary state). However, this first protocol paved the way for further investigation for a self-stabilizing solution. In the next section, we present our first self-stabilizing approach to this problem, namely, a snap-stabilizing prefix tree maintenance protocol.

5.2 Snap-stabilizing Prefix Tree Maintenance

This section presents a snap-stabilizing protocol for the maintenance of a prefix tree in a distributed environment. This section is divided as follows: we first recall, in Section 5.2.1, the definition of PGCP tree and introduce a *weaker* version of this structure (an intermediate state between *any tree* and a PGCP tree). In the same subsection, we define the theoretical model of the protocol and formally introduce *snap-stabilization*, *i.e.,* the property, for a defined protocol, to always behave as specified. Then, in Section 5.2.2 we present the repair algorithm we designed and give its correctness proof and discuss its worst case complexities. We also show some simulation results allowing to have a better idea of its average performance, in Section 5.2.3.

5.2.1 Preliminaries

In this section, we first present the distributed system model used in the design of our algorithm. Then, we recall the concept of snap-stabilization and specify the distributed data structures considered.

Distributed System

The distributed algorithm presented in this section assumes, similarly as in the previous section, an underlying physical network with a set of asynchronous *physical* nodes (*processors*, referred to as *peers*) with distinct IDs, communicating by message passing. Any peer P_1 can communicate with any peer P_2, provided P_1 knows the ID of P_2 (ignoring physical routing details). Each peer maintains one or more *logical* nodes of a distributed *logical* PGCP tree. In other words, each *logical* node (or simply *node*) of the tree is a process running on a peer. Our algorithm is run inside all these (*logical*) nodes. Recall that the tree topology is susceptible to changes during its reconstruction.

In order to simplify the design, proofs, and complexity analysis of our algorithm, we use the theoretical formal *state model* introduced in [53]. We apply this model to logical nodes (or simply, nodes) only. The message exchanges are modeled by the ability of a node to read the variables of some other nodes, henceforth referred to as its neighbors. A node can only write to its own variables. Each action is of the following form: $< label >:: < guard > \rightarrow < statement >$. The guard of an action in the program of p is a boolean expression involving the variables of p and its neighbors. The statement of an action of p updates one or more variables of p. An action can be executed only if its guard evaluates to true. We assume that the actions are atomically executed, meaning the evaluation of a guard and the execution of the corresponding statement of an action, if executed, are done in one atomic step.

The *state* of a node is defined by the values of its variables. The *state* of a system is a product of the states of all nodes. In the sequel, we refer to the state of a node and of the system as a *state* and a *configuration*, respectively. Let $\mapsto$ be a relation on $\mathcal{C}$, the set of all possible configurations of the system. A *computation* of a protocol $\mathcal{P}$ is a *maximal* sequence of configurations $e = (\gamma_0, \gamma_1, ..., \gamma_i, \gamma_{i+1}, ...)$, such that for $i \geq 0, \gamma_i \mapsto \gamma_{i+1}$ (a single *computation step*) if γ_{i+1} exists, or γ_i is a terminal configuration.

A process p is said to be *enabled* in γ ($\gamma \in \mathcal{C}$) if there exists at least an action A such that the guard of A is true in γ. We consider that any enabled node p is *neutralized* in the computation step $\gamma_i \mapsto \gamma_{i+1}$ if p is enabled in γ_i and not enabled in γ_{i+1}, but does not execute any action between these two configurations (the neutralization of a node represents the following situation: At least one neighbor of p changes its state between γ_i and γ_{i+1}, and this change effectively made the guard of all actions of p false.) We assume an *unfair and distributed daemon*. The *unfairness* means that even if a process p is continuously enabled, then p may never be chosen by the daemon unless p is the only enabled node. The *distributed* daemon implies that during a computation step, if one or more nodes are enabled, then the daemon chooses at least one (possibly more) of these enabled nodes to execute an action.

In order to compute the time complexity, we use the definition of *round*. This definition captures the execution rate of the slowest node in any computation. The set of all possible computations of $\mathcal{P}$ is denoted as $\mathcal{E}$. The set of possible computations of $\mathcal{P}$ starting with a given configuration $\alpha \in \mathcal{C}$ is denoted as $\mathcal{E}_\alpha$. Given a computation e ($e \in \mathcal{E}$), the *first round* of e (let us call it e') is the minimal prefix of e containing the execution of one action of the protocol or the neutralization of every enabled node from the first configuration. Let e'' be the suffix of e, *i.e.,* $e = e'e''$. Then *second*

round of e is the first round of e'', and so on.

Snap-Stabilization

Let $\mathcal{X}$ be a set. $x \vdash P$ means that an element $x \in \mathcal{X}$ satisfies the predicate P defined on the set $\mathcal{X}$.

Definition 2 (Snap-stabilization)**.** *The protocol $\mathcal{P}$ is snap-stabilizing for the specification $\mathcal{SP}_{\mathcal{P}}$ on $\mathcal{E}$ if and only if the following condition holds: $\forall \alpha \in \mathcal{C} : \ \forall e \in \mathcal{E}_\alpha :: \ e \vdash \mathcal{SP}_{\mathcal{P}}$.*

A Relaxed Form of PGCP Tree: the Prefix Heap

Recall Definition 1, Page 45, of a PGCP tree. In the design of our protocol, we need a *relaxed* form of the PGCP Tree. We call it a *Prefix Heap* and define it as follows:

Definition 3 (PrefixHeap)**.** *A PrefixHeap is a labeled rooted tree such that each node label is the proper greatest common prefix of all its children labels.*

The following proposition is a link between a Prefix heap and a PGCP tree and directly follows from definitions 1 and 3:

Proposition 1. *Let T be a PrefixHeap. If for any node x of T (l_x denotes the label of x) which is not a leaf node, the three following conditions are true for every pair of x's children (y, z) ($y \neq z$ and l_y, l_z denote the respective labels), then $\forall p, \forall c_1, c_2 \in C_p, l_p = PGCP(l_{c_1}, l_{c_2})$ and T is a PGCP tree:*

$$\begin{cases} (1) & l_y \neq l_z \\ (2) & l_y(\textit{resp. } l_z) \textit{ is not a prefix of } l_z(\textit{resp. } l_y) \\ (3) & |GCP(l_y, l_z)| = |l_x| \end{cases}$$

The difference between the two structures is illustrated in Figures 5.2 and 5.3. On Figure 5.2, one of the possible "heaps on the prefix" starting from the `DGEMM`, `DTRSM`, and `DTRMM` keys is shown. Figure 5.3 gives the unique PGCP tree built with these three keys.

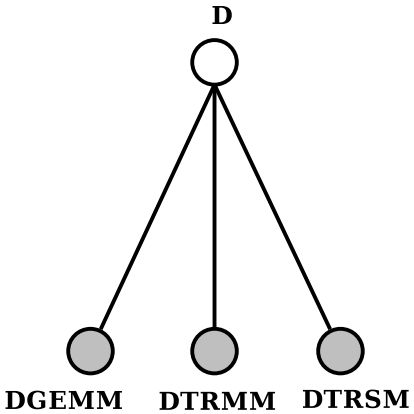


Figure 5.2: A Prefix Heap.

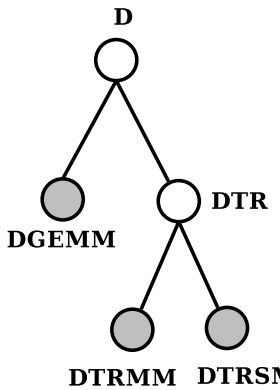


Figure 5.3: The PGCP Tree.

5.2.2 Snap-Stabilizing PGCP Tree

In this subsection, we present the snap-stabilizing PGCP tree maintenance. We provide a detailed explanation of how the algorithm works from initialization until the labels are arranged in the tree such that it becomes a PGCP tree. Next, the proof of correctness and snap-stabilization, and a discussion about the complexities are given.

The algorithm

The code of our snap-stabilizing solution is shown in algorithms 8 and 9, Page 104. We assume that initially, there exists a labeled rooted tree spanning the network. Every node p maintains a finite set of children $C_p = \{c_1, \ldots, c_k\}$, which contains the addresses of its children in the tree. Each node p is able to know the address of its parent using f_p, f_p being the node q such that q considers p as a child. [2] Initially, this relationship is well established. So, each node p can locally determine if it is either (1) the single `root` of the spanning tree (f_p is unspecified), (2) an *internal* node (f_p is specified and $C_p \neq \emptyset$), or (3) a *leaf* node ($C_p = \emptyset$). In the sequel, we denote the set of nodes in the tree rooted at p as T_p (hereafter, also called the *tree* T_p) and the *height* of the tree rooted at p as $h(T_p)$.

Each node p holds a label l_p and a state S_p in $\{I, B, H\}$[3]—stand for $Idle$, $Broadcast$, and $Heapified$, respectively. The algorithm uses two basic functions to create and delete nodes from the tree. The **NEWNODE**$(lbl, st, chldn)$ function is a process that creates a new node labeled by lbl, whose initial state is st and with a set of children initialized with $chldn$[4]. Once the new node created by this function is integrated to a set of children, the f_p macro will ensure its parent to be correctly set. Finally, the same f_p macro will set the parent variable of nodes in $chldn$[5]. The **DESTROY**(p) function is called to stop the process of a given node, (its reference should have been previously deleted from any other node).

The basic idea of the algorithm is derived from the fast version of the snap-stabilizing PIF in [34] and runs in three phases: The `root` initiates the first phase, called the *Broadcast* phase, by executing Action $InitBroadcast$. All the internal nodes in the tree participate in this phase by forwarding the broadcast message to their descendants — Action $ForwardBroadcast$. Once the broadcast phase reaches the leaves, they initiate the second phase of our scheme, called the *heapify* phase, by executing Action $InitHeap$.

During the heapify phase, a *PrefixHeap* is built — refer to Definition 3. We also ensure in this phase that for every node p, p is a single node in T_p with a value equal to l_p. The heapify phase is computed using Procedure **HEAPIFY**(), executed by all the internal nodes — Actions $BackwardHeap$. The heapify phase eventually reaches the `root` which also executes Procedure **HEAPIFY**() and initiates the third and last phase of our scheme, called the *Repair* phase — Action $InitRepair$. The aim of the *Repair* phase is to correct the two following problems that can occur in the *PrefixHeap*. First, even if no node in T_p has the same label as p, the same label may exist in other branches of the tree; Second, if each node is the greatest common prefix of its children labels, it is not necessarily the greatest common prefix of any pairs of its children labels.

[2] In a real P2P network, the relationship child/parent is easily preserved by exchanging messages between a child node and its parent.

[3] To ease the reading of the algorithm, we assume that $S_p \in \{I, B\}$ (respectively, $\{I, H\}$) if p is the `root` (resp., p is a leaf). We could easily avoid this assumption by adding the following guarded action for the `root` (resp. leaf) node: $S_p = H$ (resp. $S_p = B$) $\longrightarrow S_p := I$. Note that this correction could occur only once.

[4] See previous chapters for its implementation

[5] Again, the implementation of such a process in message passing can simply rely on one message sent by the newly created node to its set of children.

The *Repair* phase works similarly as the *Broadcast* phase but has its own semantic. The `root` and internal nodes execute Procedure **REPAIR**() starting from the `root` toward the leaves — Actions $InitRepair$ and $ForwardRepair$. During this phase, for each node p, four cases can happen (refer to Algorithm 9):

1. Several children of p have the same label. Then, all the children with the same label are merged into a single child — Lines 11.02 to 11.07;
2. The labels of some children of p are prefixed with the label of some of its siblings. In that case, the addresses of the prefixed children are moved into the corresponding sibling — Lines 11.08 to 11.12;
3. The labels of some children of p are prefixed with a label which does not exist among their siblings and which are longer than the label of p. Then, for each set of children with the same prefix, p builds a new node with the corresponding prefix label and the corresponding subset of nodes as children — Lines 11.13 to 11.16.
4. If none of the previous three cases appear, nothing is done.

Finally, the *Repair* phase ends at leaf nodes by executing Action $EndRepair$. This indicates the end of the PGCP tree construction. Note that since we are considering self-stabilizing systems, the internal nodes need to correct abnormal situations due to the unpredictable initial configuration. The unique abnormal situation which could avoid the normal progress of the three phases of our scheme is the following: An internal node p is in state B (done with its broadcast phase) but its parent f_p is in state H or I, indicating that it is done executing its heapify phase or it is Idle, respectively. In that case, p executes the action $ErrorCorrection$, in the worst case, pushing down T_p the abnormal broadcast phase until reaching the leaf nodes of T_p. This guarantees the liveness of the protocol despite unpredictable initial configurations of the system.

Correctness proof

In this section we show that the algorithm described is a snap-stabilizing PGCP tree algorithm. The complexities are also discussed.

Remark 2. *To prove that an algorithm provides a snap-stabilizing PGCP tree algorithm, we need to show that the algorithm satisfies the following two properties:* (1) *starting from any configuration, the* `root` *eventually executes an initialization action;* (2) *Any execution, starting from this action, builds a PGCP tree.*

Let us first consider the algorithm by ignoring the two procedures **HEAPIFY**() and **REPAIR**(). In that case, the algorithm is very similar to the snap-stabilizing PIF in [34]. The only difference between both algorithms is in the third phase. In Algorithm 8, the third phase is initiated by the `root` only, *after* the heapify phase terminated only, whereas in [34], the third phase can be initiated by any node once itself and its parent are done with the second phase. That means that with the solution in [34], both the second and the third phase can run concurrently. That would be the case with Algorithm 8 if the guard of Action $ForwardRepair$ has been as follows: $S_p = H \wedge S_{f_p} \in \{H, I\} \wedge (\forall c \in C_p : \ S_c \in \{H, I\})$.

However, it follows from the proofs in [34] that the behavior imposed by our solution is a particular behavior of the snap-stabilizing PIF algorithm. This behavior happens when all the nodes are slow

to execute the action corresponding to the third phase. Since the algorithm in [34] works with an unfair daemon, the algorithm ensures that, eventually, the `root` initiates the third phase, leading the system to behave as Algorithm 8. Therefore, ignoring the effects of the two procedures **HEAPIFY**() and **REPAIR**() on the tree topology, the proof of snap-stabilization in [34] is also valid with our algorithm.

Considering the two procedures **HEAPIFY**() and **REPAIR**() again, since $\forall p$, the set C_p is finite, it directly follows from the code of the two procedures in Algorithm 9 that every execution of the procedures **HEAPIFY**() or **REPAIR**() is finite.

It follows from the above discussion:

Lemma 3. *Starting from any configuration, the* `root` *node can execute Action InitBroadcost in a finite time even if the daemon is unfair.*

As a corollary of Lemma 3, the first condition of Remark 2 holds. Also, this shows that every computation initiated by the `root` eventually terminates. It remains to show that the second condition of Remark 2 also holds for any node p.

Lemma 4. *After the execution of procedure* **HEAPIFY***() by a node p, T_p is a PrefixHeap.*

Proof. We prove this by induction on $h(T_p)$. Since procedure **HEAPIFY**() cannot be executed by a leaf node, we consider $h(T_p) \geq 1$. Note that the code of the function **HEAPIFY**() is triggered only for nodes whose label is **not** the PGCP of its children' labels. In other words, the function executes something after the test of Line 10.02 only if it is required.

1. Let $h(T_p)$ be equal to 1. So, all the children of p are leaves. Executing Lines 10.03 to 10.04, p is as a new child, itself a leaf node, labeled with l_p, while l_p contains the greatest common prefix (GCP) of all its children. After the execution of Lines 10.05 to 10.08, p contains no child c such that $l_c = l_p$. Thus, l_p is a PGCP of all its children labels.

2. Assume that the lemma statement is true for any p such that $h(T_p) \leq k$ where $k \geq 1$. We will now show that the statement is also true for any p such that $h(T_p) = k + 1$. By assumption, the lemma statement is true for all the children of p, *i.e.,* $\forall c \in C_p$, l_c is a proper prefix of any label in T_c, and l_c is the PGCP of all nodes in C_c. So, after executing procedure **HEAPIFY**(), following the same reasoning as in Case 1, l_p is a PGCP of all its children, and since themselves are the root of a PrefixHeap, for every $c \in C_p$, l_p is also a proper prefix of any label in T_c. Hence, the lemma statement is also true for p.

□

Corollary 1. *After the system executed a complete Heapify phase, the whole tree (i.e., $T_{\texttt{root}}$) is a PrefixHeap.*

Lemma 5. *If $T_{\texttt{root}}$ is a PrefixHeap, then after the execution of Procedure* **REPAIR***() by any node p such that $h(T_p) \geq 1$, for every pair $(c_1, c_2) \in C_p$, $l_p = PGCP(c_1, c_2)$.*

Proof. Consider first that p is the `root`. By executing Lines 11.02 to 11.07, if there exists a pair $(c_1, c_2) \in C_p$ such that $l_{c_1} = l_{c_2}$, then p replaces all the nodes with the same label c_1 by a unique node gathering all their child sets together. So, after the execution of Lines 11.02 to 11.07, for every pair $(c_1, c_2) \in C_p$ $(c_1 \neq c_2)$, $l_{c_1} \neq l_{c_2}$. Following the same reasoning, after the execution of Lines 11.08

to 11.16, for every pair $(c_1, c_2) \in C_p$ ($c_1 \neq c_2$), l_{c_1} (resp. l_{c_2}) is not a prefix of l_{c_2} (resp. l_{c_1}), and $|GCP(l_{c_1}, l_{c_2})| = |l_p|$. Thus, by Proposition 1, if p is the `root`, the lemma holds.

Consider now that p is not the `root`. Since Procedure **REPAIR**() modifies the set C_p only, following the same reasoning as for the `root`, after the execution of Lines 11.02 to 11.16, by Proposition 1 again, the lemma holds for p. □

Corollary 2. *If $T_{\texttt{root}}$ is a PrefixHeap, then after the system executed a complete Repair phase, the whole tree is a PGCP tree.*

Proof. By induction of Lemma 5 on every node of the path from the `root` to each leaf node. □

From Corollaries 1 and 2, and the fact that after the `root` executed Action *InitBroadcast*, the three phases *Broadcast*, $Heapify$, and *Repair* proceed one after another, we can claim the following result:

Theorem 2. *Running under any daemon, Algorithm 8 and Algorithm 9 provide a snap-stabilizing Proper Greatest Common Prefix Tree construction.*

This protocol is *snap-stabilizing seen from the root of the tree*, as the root needs to launch the protocol only once to ensure that the tree will be a consistent PGCP tree at the end of the computation. It is not the case seen from the client, who, if sending a request to the tree, may connect the tree when it is faulty, and within which the repair protocol is still running, or even was not launched. In other words, we consider here that the discovery process and maintenance process are distinct and run concurrently.

Complexities Discussion

Theorem 3. *The time complexity for the PGCP tree construction is $O(h + h')$ rounds. In the worst case, the construction requires an $O(n)$ extra space complexity, $O(n)$ rounds and $O(n^2 \times E)$ operations, where n is the number of nodes of the tree and E the cost of the underlying system returning reference of processes.*

Proof. By similarity with the PIF, we can easily establish that the *broadcast* phase reaches all leaf nodes in $O(h)$ rounds, where h is the height of the tree when Action *InitBroadcast* is triggered. We also easily see that the *heapify* phase reaches the root in $O(h)$. Note that the time complexity of the **DESTROY**() procedure can be assumed constant as no other node has to be destroyed (its children have all been adopted by another node before it is destroyed.) During the *repair* phase, the number of rounds required to reach all leaf nodes of the repaired tree is clearly $O(h')$, where h' is the height of the final tree (one execution of the **REPAIR**() procedure increases the depth by 0 or 1).

The extra space required on a node p depends directly on the number of children of p, which can clearly not be higher than $n - 1 = O(n)$. This happens when the tree is a star graph, every node except the `root` being a child of the `root`.

Dealing with the time complexity, the highest value for h and h' is n (in the case the tree is a chain), what establishes that the worst case requires $O(n)$ rounds. It remains to give the worst complexity of one round. The cost of the **NEWNODE** function is mainly the cost of getting the reference through the underlying system, which we denote E. The remainder of the **NEWNODE** function is assumed atomic (see below for a discussion). Then, independently of this function, as we easily see from the code, the time complexity of Procedure **REPAIR**() on a node p depends on $|C_p|$. Recall that, once this procedure has been executed on a node p, it is executed in parallel on each

$c \in C_p$. After the execution of **REPAIR**() on p, we easily see that $\forall c \in C_p, |C_c| \leq |C_p| - 1$. This bound is reached when **REPAIR**() on p moves $|C_p| - 1$ children of p under one single node in C_p. If this scenario repeats at each level of the tree, no parallelism is achieved during the whole *repair* phase and the i^{th} round is made of $a \times (|C_{\texttt{root}}| - i)$ operations where a is a constant factor. As we previously established, $|C_{\texttt{root}}| \leq n - 1$. Finally, the number of operations in the worst case is:

$$[a \times (n-1) + a \times (n-2) + \ldots + a] \times E = O(n^2 \times E)$$

□

As in Chapter 3, we have here assumed the presence of a cache system in the underlying system providing peers' references, what can lead to $E = O(1)$. In addition, implementing the f_p macro in the case where some nodes need to update their parent after the creation of a new node can be done through a message sent by the new node to all its children in parallel, justifying a constant time on this part.

In the following section, we expose simulation results showing that, in real settings, the worst case is far to be reached, both in terms of latency and extra space.

5.2.3 Simulation Results

To better capture what we can expect from the behavior of the snap-stabilizing PGCP tree, we simulated the algorithm again using data sets which reflect the use of computational platforms. The simulator is written in Python and contains the three following parts:

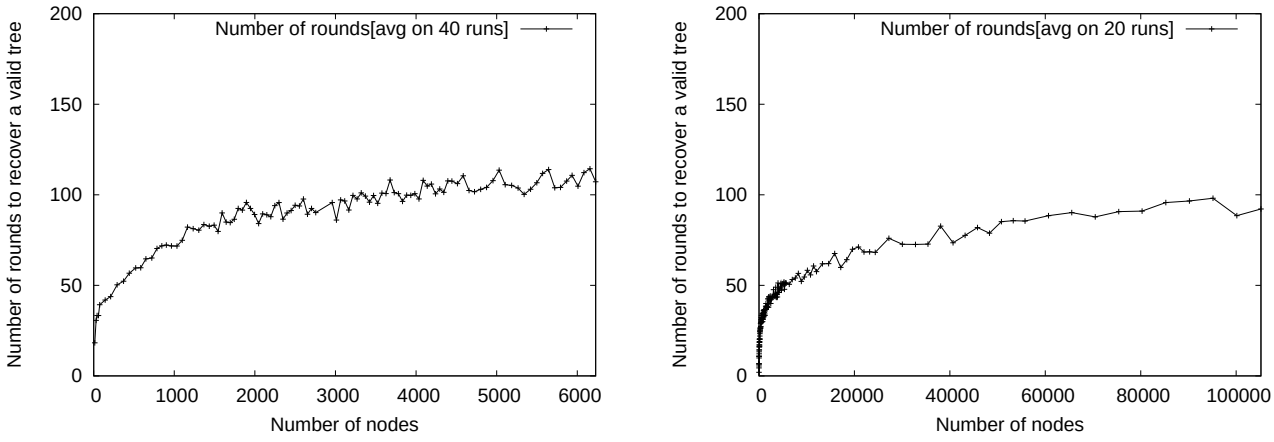
1. It creates the tree with a set of labels of basic computational services commonly used in computation grids such as the names of routines of linear algebra libraries, the names of operating systems, the processors used in today's clusters and the nodes' addresses. The number of keys is up to 5200, creating trees up to 6228 nodes. For instance, inserting two labels `DTRSM` and `DTRMM` results in a tree whose root (common parent of `DTRSM` and `DTRMM`) is labeled by `DTR`.

2. It destroys the tree by moving subtrees, randomly. This is achieved by modifying the parent of another randomly picked node, moving it from the set of children of its parent to the set of children of a randomly chosen node. This operation is repeated on up to $n/2$ nodes (meaning that approximately $n/2$ nodes are connected to a wrong parent). Nodes are initialized in State I. (However, nodes could have been initialized in any state, since the algorithm would have immediately changed their State to I.)

3. It executes the algorithm by testing for every node if its state and those of its neighbors satisfy the guard of some action in the algorithm, in which case the statement of the action is executed on the node. This whole process is a **round** (see Section 5.2.1) within which all nodes are synchronized (any enabled node triggers the action corresponding to its state) and is repeated until the tree is in a configuration where all nodes are in state I again.

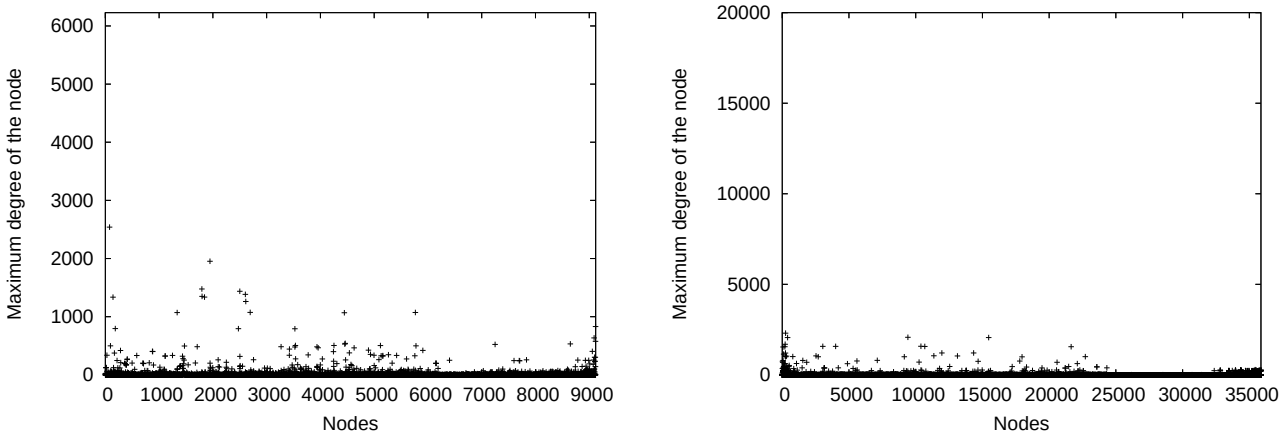
   Recall that the protocol always behaves according to its specification, *i.e., executions* are always valid. Here, we focus on the *configurations* of the system and in the number of *rounds* required to have a tree in which all nodes are in state I again, and thus to have a valid PGCP tree.

We have first collected results on the latency of the algorithm. Figure 5.4(a) gives the average number of rounds required to have a valid PGCP tree, starting from 40 different arbitrary settings of



(*a*) *Grid-oriented* keys. (*b*) Random keys.

Figure 5.4: Snap-stabilizing protocol: latency results.



(*a*) *Grid-oriented* keys. (*b*) Random keys.

Figure 5.5: Snap-stabilizing protocol: extra space results.

the tree. The tree size ranges between 2 and 6228. We observe that the number of rounds required by the algorithm has a logarithmic behavior (far from the linear worst case). It clearly scales according to the height of the tree, thus confirming the average complexity of the algorithm and its good scalability under more actual settings. Figure 5.4(*b*) gives similar results, but with a tree whose size is up to more than 100000, built with randomly generated keys of length between 1 and 25.

We have also collected results on the extra space required on each node. Since the tree topology undergoes changes during the reconstruction, degrees of nodes also dynamically change as nodes are created, destroyed, merged or moved. Figure 5.5(*a*) and (*b*) shows the highest degree of nodes, *i.e.,* the real extra space required on each node, including those of nodes created and/or destroyed during the reconstruction. Using *grid* keys, the final tree size is 6228; the total number of nodes, including temporary nodes (created and destroyed later), is 9120. We compute the maximum temporary extra space required on one node (the maximum number of neighbors it had to manage at a given time).

The experiment shows that the highest of maximum degree of all nodes is 2540, and most of maximum degrees are very low (less than 50). This can be partly explained by the fact that the deeper a node is, the smaller is its degree. In other terms, during a breadth-first traversal of the tree, the topology quickly enlarges close to the root and then its breadth grows slowly on the way to the leaf nodes. Using randomly generated keys leads to a similar result. More generally, this simulation shows that the worst case is far to be reached and that only few nodes will require a large extra space.

Snap-Stabilization Enabled in Peer-to-Peer Prefix Trees. This second protocol is snap-stabilizing and thus optimal in terms of convergence time. But, to be able to prove this property, we modeled our system in the state model. As a consequence, the correctness of the algorithm has been established only in this model, and can not be applied on a real message-passing platform in a straightforward manner. Moreover, this protocol assumes the logical topology is always a rooted connected tree.

Addressing the issue of practicability, our last protocol is a complete message-passing self-stabilizing protocol to maintain a prefix tree on a real platform. Beyond the proof of self-stabilization of this protocol, we study further the actual effectiveness of self-stabilization in our particular context.

5.3 Self-stabilizing Message-Passing Prefix Tree Maintenance

In this section, we propose a self-stabilizing protocol to maintain a prefix tree in a message passing model designed for P2P architectures. Our algorithm does not require a fixed root node and works with any arbitrary initial configuration (possibly disconnected) of the tree topology. As before, we need a service to keep a knowledge of the physical network (*weakly-connected continuously bootstrap mechanism*). To this end, we use a model similar to the one proposed in [76]. Being written in a *realistic* model, the proposed protocol can be implemented on any platform that supports message passing and basic services available in most peer-to-peer systems. We give a formal proof of correctness of our protocol and some simulation results to study the scalability of the protocol as well as its efficiency in keeping the architecture available for clients even under high failure rate. In Section 5.3.1, we give the model in which our protocol is designed, and the data structures it maintains. The protocol is given in Section 5.3.2, followed by its proof in Section 5.3.3. Simulation results are given in Section 5.3.4.

5.3.1 Preliminaries

The Network. As previously, we assume A P2P network consisting of a set of asynchronous processors with distinct ids. The processors communicate by exchanging messages. Any processor P_1 can communicate with another processor P_2 provided P_1 knows the *id* of P_2. We abstract the details of the actual routing. Henceforth, we use the word *peer* to refer to a processor.

The Logical Tree. Each peer maintains a part of our indexing system *i.e.,* some *logical* nodes (that we want to group into a proper greatest prefix tree). Keep in mind that a (*logical*) node is implemented as a process, which runs on a peer. In other words, a process implements the notion of node.

We now go a bit further in the description of our system, trying to avoid any misunderstandings. As we said, each node has a label. In a correct configuration (that we have defined in Definition 1, Page 45), each node label is unique. (Only one node is responsible for all services may share a

common label.) However, initially, when the system is not yet stabilized, the structure may contain errors and, as an example, multiple nodes sharing the same label. Thus, we cannot use labels to identify the nodes. We chose to identify nodes by the process implementing it. A process is identified by a unique combination of the peer running it and a port number. Our protocol maintaining the prefix tree is run on every process. *Nodes* and *Processes* are basically two different views of the same thing. In the remainder, we use these terms interchangeably. If our point is more related to the tree structure, we will use the word *node*. If our point is more related to basic entities running on the network, we will use the word *process*. Recall that the tree topology is again susceptible to changes during its reconstruction. We assume the presence of a service able to return process references. This *process discovery* service is similar to the one we used before and similar to the one described in [76]. Any process of the system can obtain any other process identifier by calling this service. To prove the correctness of the algorithm, we assume that a finite number of queries to this service is enough to collect the identifiers of all processes in the system. The service provides the following two primitives: GetRunningProcess() returns the identifier of a randomly chosen process, and GetNewProcess() creates a new process (without setting its parameters yet, see later the InitProcess() function for this purpose) and returns its identifier. The communication between processes is carried out by exchanging messages. A process p is able to communicate with a process q, if and only if p knows the id of q. We assume that a copy of every message sent by p to q is eventually received by q, unless q terminated (crash, kill). The message delay is finite but not bounded. Messages arrive in the order in which they were sent (FIFO), and as long as a message is not processed by the receiving process, we assume that it is in transit.

Self-Stabilization. Define a *transition system* as a triple $\mathcal{S} = (\mathcal{C}, \mapsto, \mathcal{I})$, where $\mathcal{C}$ is a set of configurations, $\mapsto$ is a binary transition relation on $\mathcal{C}$, and $\mathcal{I} \subset \mathcal{C}$ is the set of initial configurations. A *configuration* is a vector with $n+1$ components, where the first n components are the state of n processes and the last one is a multi-set of messages in transit in m links. We define an *execution* of $\mathcal{S}$ as a maximal sequence $\mathcal{E} = (\gamma_0, \gamma_1, \gamma_2, ..., \gamma_i, \gamma_{i+1}, ...)$, where $\gamma_0 \in \mathcal{I}$ and for $i \geq 0, \gamma_i \mapsto \gamma_{i+1}$. A predicate Π on $\mathcal{C}$, the set of system configurations, is *closed* for a transition system S if and only if every configuration of an execution e that starts in a configuration satisfying Π also satisfies Π. A transition system S is *self-stabilizing* with respect to a predicate Π if and only if Π is closed for S and for every execution e of S, there exists a configuration of e for which Π is true.

5.3.2 Protocol

The proposed algorithm builds a PGCP tree starting from an arbitrary logical network where each node is labeled.

Communications. The protocol assumes the existence of an underlying *self-stabilizing end-to-end communication* (SSEE) protocol. Both layers communicate using **send/receive** primitives over FIFO message queues. The "**send**(<m>, q)" primitive sends the message m to the node q. It always terminates and, either the recipient q is alive and the message m is queued on q, and will be processed later, or q crashed and is no longer available and the message is lost. The implementation of Protocol SSEE is beyond our scope. Please refer to [23, 56].

Notations. Every process p (we use p to denote the *id* of p and the address used by other processes to communicate with p) has a label l_p. Denote by $\widehat{p}$, the pair (p, l_p). Recall that, $\widehat{p} = \widehat{p'}$ is equivalent

to $(p = p') \wedge (l_p = l_{p'})$. Node p also maintains the id and label of its parent into $\widehat{f_p}$ and its children into the finite set $\widehat{C_p}$. Note that $\widehat{C_p}$, $\hat{p}$ and $\widehat{f_p}$ are variables, C_p is the result of a macro that extracts the first element of every pairs in $\widehat{C_p}$.

Heartbeat. To deal with crash failures, to maintain the topological information, and to maintain the status of processes (they terminated or not) in our protocol, we assume the presence of an underlying *heartbeat* protocol. We assume that any node that does not receive news from one child or parent during a bounded time (implemented by using a TimeOut action in the algorithm), removes the node from its neighborhood set. Note that when deleting a given child $q \in C_p$, all the data associated with q is deleted.

Processes Discovery. The function GETEPSILON() returns the identifier of a random node labeled by the empty word ϵ. It relies on the *process discovery* service previously described. Basically, it calls GETNEWPROCESS() and checks if the id returned is an ϵ-process, *i.e.*, a process labeled by ϵ. Since we assume that a finite number of calls to the *process discovery* service is enough to get all identifiers of alive processes, the GETEPSILON() function also returns every ϵ-process in a finite time. The INITPROCESS(lbl, f, C) function initializes a process on the local node and sets the label with lbl, the parent with f and the set of children with C.

The Algorithm

The rules of the protocol, *i.e.,* the *periodic* rule, periodically runs on each node as detailed in Algorithm 10, Page 105, and the *upon receipt* rules, detailed in Algorithm 11, Page 106, triggered on the receipt of a message, are atomic.

Each node p periodically initiates the action described by Algorithm 10. Process p begins by eliminating the cases where p is either a parent or a child of itself, what could lead in configurations with cycles, that are hard to systematically eliminate — see Lines 12.02-12.03.

Lines 12.04-12.13 deal with parent maintenance. These lines ensure that eventually, there will be one and only one root, *i.e.,* only one node p eventually satisfies $f_p = \perp$. To achieve this, the possible root nodes merge. Let us consider a root node p to detail this part of the algorithm. There are two possible situations:

1. If the label of p is ϵ, p tries to connect to another node q, also labeled ϵ. q then becomes a child of p (Line 12.08). p informs q that its parent changed using UPDATEPARENT message. Upon receipt of that message, q updates its parent variable (Lines 16.01-16.03 of Algorithm 11). Since p and q are labeled identially, they will merge (the merging is explained below), thus reducing the number of roots by one.

2. If p is not labeled by ϵ, a new node labeled ϵ is artificially created as the parent of p. This new node executes the periodic rule satisfying the previous case.

Lines 12.15-12.26 deal with children maintenance to make sure that eventually, every set of children satisfies Definition 1. This phase consists of three parts.

1. We eliminate cases where the set of children of p contains a node q whose label is the label of p by initiating the merging of p and q. p sends a MERGE message to q (Lines 12.15-12.16). First, upon receipt of the MERGE message, q informs its children that their new parent is

their current grandparent through GRANDPARENT messages. Upon receipt of this message, the children of q change their parent from q to p. To ensure a good synchronization, q waits until all its children have been accepted by p as children, *i.e.,* waits for the GFDONE message. q finally informs p that the merging process has finished by sending the MDONE message, and terminates (Lines 18.01-21.03).

2. We eliminate cases where a child q_1 prefixes another child q_2. So, the proper greatest common prefix of the labels of q_1 and q_2 is equal to the label of q_1. But, the greatest common prefix, by Definition 1 must be the label of p. A contradiction (Line 12.17). q_2 then becomes the child of q_1 (Lines 12.17-12.19).

3. We check that there is no pair (q_1, q_2) in its set of children such that the greatest common prefix g of their labels is greater than its own label (Lines 12.20-12.26). In this case, a new node must be created. This node, labeled by g, will be the child of p and the common parent of q_1 and q_2.

The purpose of Lines 12.27-12.28 is for p to check the validity of its parent. Upon receipt of the PARENT message, the parent of p decides whether p is its child depending on their labels, and informs p of the result (Lines 13.01-13.06). It uses a CHILD message to indicate that it considers p as its child. Otherwise, it sends an ORPHAN message. Lines 14.01-15.03 detail the receipt of these messages. Upon receipt of CHILD, p updates the label of its parent. Upon receipt of ORPHAN, it becomes a root and will execute the periodic rule as we discussed before.

5.3.3 Proof of Stabilization

In many distributed systems where failures (or topology changes) can occur — *e.g.,* peer-to-peer networks—, the failure frequency must be assumed to be "*low enough*" to have "*enough time*" to achieve the intended goal of a given protocol. For instance, no one could reasonably claim that a peer-to-peer protocol, like a Distributed Hash Table, works under the assumption that peers can crash "so fast" that none of them has time to send even one message. Since the property of self-stabilization guarantees the convergence of the system to its intended behavior in finite time, it can model the ability for the system to recover from transient failures, and also, to tolerate changes of its topology brought about failures or repairs of its components [126]. Stabilization is usually proven under the assumption that no failure occurs from the beginning. In other words, if failures occur, then they occur *before* the first considered system configuration. Therefore, the above ability makes sense assuming similar reasonable assumption as above, *i.e.,* (i) the frequency of fault occurrence is not too high, and (ii) the time between two occurrences of faults is higher than the time required to recover from a fault.

In light of the above assumptions, while proving the correctness of stabilizing algorithms (especially, their convergence property), we assume that no fault occurs during the convergence period. In other words, we only need to show the convergence of the protocol after the last fault occurs.

Following the above discussion, to prove the correctness of the proposed protocol, we consider a suffix of an execution starting after all crashes have taken place, i.e., in this particular execution segment, no crashes occur. Let P be the set of alive processes. Every "**send**($<m>$, q)" executed by a process $p \in P$ terminates and when this happens, either m is received by q or $q \notin P$.

A configuration γ satisfies Predicate Π_1 if and only if, assuming that a process $p \in P$ infinitely often sends a message to a process $q \in P$ $(q \neq p)$, both conditions are true in every execution e

starting from γ: (1). Safety: the sequence of messages received by q is a prefix of the sequence of messages sent by p; (2). Liveness: q receives a message infinitely often.

The following lemma directly follows from the fact that we assume an underlying self-stabilizing end-to-end communication protocol:

Lemma 6. *The system is self-stabilizing with respect to* Π_1.

Corollary 3. *Every message received by a process in* P *in a configuration satisfying* Π_1 *has been sent by another process in* P.

Lemma 7. *Starting from a configuration satisfying* Π_1, *every process in* P *executes Lines 12.01-12.28 infinitely often.*

Proof. The set $\widehat{C_p}$ is finite and the loop is executed atomically (no message receipt can interrupt the execution of the loop). Moreover, each execution of **send** terminates. So, none of the three "while loop" (Lines 12.15-12.26) may loop forever. The lemma follows. □

Corollary 4. *Starting from a configuration satisfying* Π_1, *the system eventually contains no process* p *such that* $f_p = p$ *or* $p \in C_p$.

Proof. Process p executes Lines 12.02 and 12.03 infinitely often. Moreover, the algorithm contains no line in which $f_p := p$ or $\widehat{C_p} := \widehat{C_p} \cup \{\widehat{p}\}$. □

We will now show that, starting from a configuration satisfying Π_1, the child set (C_p) of each process $p \in P$ eventually contains no child Id q such that $q \notin P$ — *i.e.,* q is alive.

Lemma 8. *Let* p *be a process in* P. *In every execution starting from a configuration* γ *satisfying* Π_1, *if there exists* $q \in C_p$ *such that* $q \notin P$, *then, eventually,* $q \notin C_p$.

Proof. It follows the assumption of the presence of an underlying *heartbeat* protocol between neighbors. □

Let Π_2 be the predicate over $\mathcal{C}$ such that $\gamma \in \mathcal{C}$ satisfies Π_2 iff $\forall p \in P$, $\forall q \in C_p$, $q \in P$.

Lemma 9. *The system is self-stabilizing with respect to* Π_2.

Proof. From Corollary 3, no process $p \in P$ can receive a message from a process $q \notin P$. So, in any execution starting from a configuration γ satisfying Π_1, no process p can add a process Id q such that $q \notin P$. p can add a process q in C_p using Lines 12.08 and 12.26 in which case q was returned by a GET*() function assumed to return ids in P. It can also add q using Line 13.03, in which case q was sent by q itself, and is thus alive. By Lemma 8, if there exists some process $p \in P$ such that C_p contains ids not in P, then each of these ids is eventually removed from C_p. Thus, eventually, $\forall p \in P$, $\forall q \in C_p$, $q \in P$. □

From now on, we do not mention P because all process references are assumed to be in P. Let Π_3 be the predicate over $\mathcal{C}$ such that $\gamma \in \mathcal{C}$ satisfies Π_3 iff in every execution starting from γ satisfying Π_2, for each process p: (1) p executed Lines 12.01-12.28 at least once, and (2) if p sent a message "PARENT?" to q, then p received the corresponding answer (a message <CHILD> or <ORPHAN>) from q. The following lemma is straightforward: from Lemma 7 and the fact that every message receipt terminates:

Lemma 10. *The system is self-stabilizing with respect to* Π_3.

Lemma 10 ensures that for every process p, each label in $\widehat{C_p}$ is correct, *i.e.,* is equal to the actual label of q. p adds or updates the child labels in three ways. First, using Line 13.03 in which case the label was sent by q itself and is thus correct. Second, by using Line 12.08 in which case q was returned by GetEpsilon() and the label is set to ϵ. Third, by using Line 12.26 in which case the label was computed by p itself and then sent to *new*. We will now show that, starting from a configuration satisfying Π_2, the child set (C_p) of each process p eventually contains no child Id q such that $l_p \notin \text{Prefixes}(l_q)$.

Lemma 11. *Let p and q be two processes. If there exists an execution starting from a configuration satisfying Π_3 containing a system transition $\gamma_t \mapsto \gamma_{t+1}$ such that $q \notin C_p$ in γ_t and $q \in C_p$ in γ_{t+1}, then $l_p \in \text{Prefixes}(l_q)$.*

Proof. To add q to C_p, p executes one of the following lines:

1. Line 12.08. In this case, $l_p = l_q = \epsilon$.
2. Line 12.26. In this case, $q = new$, and $l_q = GCP(l_{q_1}, l_{q_2})$, where both l_{q_1} and l_{q_2} are prefixed by l_p.
3. Line 13.03. This line is executed only if $l_p \in \text{Prefixes}(q)$ — see Line 13.02.

□

Lemma 12. *Let γ be a configuration satisfying Π_3. Let p and q be a pair of processes such that, in γ, $q \in C_p$. If there exists an execution e starting from γ such that $q \in C_p$ forever, then $l_p \in \text{Prefixes}(l_q)$.*

Proof. Assume by contradiction that there exists e starting from γ such that $q \in C_p$ forever, and $l_p \notin \text{Prefixes}(l_q)$. There are two cases to consider:

1. There exists a configuration $\gamma' \in e$ such that $f_q \neq p$ forever ($f_q \neq p$ in every execution starting from γ'). In that case, assuming the presence of an underlying *heartbeat* protocol between neighbors, p will not receive heartbeats from q and eventually remove it from the set of its children. A contradiction.
2. $f_q = p$ infinitely often. So, q sends Parent? to p infinitely often. Upon receipt of such a message, p removes q from C_p—Line 13.06. A contradiction.

□

Let Π_4 be the predicate over $\mathcal{C}$ such that $\gamma \in \mathcal{C}$ satisfies Π_4 iff given two processes p, q, if $q \in C_p$ in γ, then $l_p \in \text{Prefixes}(l_q)$.

Lemma 13. *The system is self-stabilizing with respect to* Π_4.

Proof. By Lemma 12, for every process p, if C_p contains q such that $l_p \notin \text{Prefixes}(l_q)$, then q is eventually removed from C_p. By Lemma 11, for every p, q can be added to C_p only if $l_p \in \text{Prefixes}(l_q)$. So, eventually, if $q \in C_p$, then $l_p \in \text{Prefixes}(l_q)$. □

It follows from Lemma 13 that, in every configuration γ satisfying Π_4, if there exists a process p and no q such that $l_p \in \text{PREFIXES}(l_q)$, then C_p is an empty set. In other words, the leaf nodes are without child forever. We will now show that there is eventually only one tree.

Lemma 14. *In every execution starting from a configuration γ satisfying Π_4, the number of times a process p sets f_p to $\perp$ is less than or equal to 1.*

Proof. Assume by contradiction that there exists an execution e starting from γ and a process p setting f_p to $\perp$ more than once. In a configuration satisfying Π_4, by Corollary 4 and Lemma 9, p can set f_p to $\perp$ upon receipt of a message ORPHAN only. So, p receives ORPHAN at least twice. After the first receipt, p executes the loop Lines 12.01-12.28. There are two cases to consider:

1. $l_p = \epsilon$. In that case, p obtains an existing "ϵ-process" q' as its parent — refer to Lines 12.05-12.09. Then, p sends UPDATEPARENT to q' that will never sends ORPHAN to p since $l_q \in \text{PREFIXES}(l_p)$.

2. $l_p \neq \epsilon$. In that case, p creates and chooses as a parent a new "ϵ-process" q. This case is similar to the first one.

□

Let ϱ be the number of processes p such that $f_p = \perp$.

Lemma 15. *In every configuration γ satisfying Π_4, if $\varrho = 0$ in γ, then ϱ is eventually greater than 0 and remains greater than 0 thereafter.*

Proof. Assume by contradiction that $\varrho = 0$ in γ and there exists an execution e starting from γ such that ϱ is equal to 0 infinitely often. There are two cases to consider:

1. $\varrho = 0$ in every configuration of e, *i.e.,* $\forall p,\ f_p \neq \perp$ in every configuration. So, no process ever receives ORPHAN. Let p be a process such that $\forall q \neq p,\ l_q \notin \text{PREFIXES}(l_p)$—*i.e.,* l_p is minimum. (Note that in every configuration satisfying Π_4, $\forall q \neq p,\ p \notin C_q$.) Upon the first receipt of PARENT? sent by p to its parent, say p', p' sends ORPHAN to p. A contradiction.

2. $\varrho = 0$ infinitely often. From Lemma 14, $\forall p \in P$, p sets f_p at most once. So, ϱ increases from 0 to a value $x \leq |P|$. Then, since we assume that $\varrho = 0$ infinitely often, it means that ϱ will then be equal to 0, eventually. And since ϱ can not increase anymore, it will remains equal to 0, which is the first case.

□

Lemma 16. *In every execution starting from a configuration γ satisfying Π_4, ϱ is eventually equal to 1.*

Proof. By Lemmas 14 and 15, in every execution from γ, there exists a configuration γ_t such that ϱ is equal to a maximum value $x \in [1, |P|]$. Assume by contradiction that there exists an execution e, a value $y \in [2, x]$, and a configuration γ'_t in e with $t' \geq t$ such that $\varrho = y$ and remains equal to y thereafter. There are two cases to consider:

1. Among the y nodes, there exists p such that $l_p \neq \epsilon$. Then, p eventually executes Lines 12.11-12.13 a new ϵ-process is created, taking p as its child. The number of roots is unchanged but, eventually, every root is labeled by ϵ.

2. The label of the y nodes is equal to ϵ. Let p be the ϵ-processes having the maximum identifier. By executing Line 12.06, p eventually chooses an ϵ-process q such that q sets f_q to p upon receipt of the message UPDATEPARENT sent by p, and the number of roots is decremented. A contradiction.

□

Let Π_5 be the predicate over $\mathcal{C}$ such that $\varrho = 1$.

Lemma 17. *The system is self-stabilizing with respect to* Π_5.

Proof. Follows from Lemmas 14, 15, and 16. □

In every configuration satisfying Π_5, there exists a single process r such that $l_r = \epsilon$ and $f_r = \perp$. In the next and last step of the proof, we show that if the parent of a process p changes, then p moves toward the leaves such that the tree eventually forms a PGCP tree.

Lemma 18. *In every execution starting from a configuration* γ *satisfying* Π_5*, if a process* p *sets* f_p *to* q*, then* $l_q \in \text{PREFIXES}(l_p)$.

Proof. In every configuration γ satisfying Π_5, a process can change f_p by executing the receipt of either a message GRANDPARENT or UPDATEPARENT, in both cases, sent by its parent. In both cases, f_p is set to q such that $l_q \in \text{PREFIXES}(l_p)$. □

Lemma 19. *In every execution starting from a configuration* γ *satisfying* Π_5*, the number of pair* p, q *such that* $l_p = l_q$ *is eventually equal to* 0.

Proof. Note that in every configuration γ satisfying Π_5, one among $\{p, q\}$ is the parent of the other. Without loss of generality, we assume that p is the parent of q. By the repeated executions of Lines 12.15-12.16 and 18.01-21.03 on each pair p, q, all the children of q eventually become the children p and q eventually disappears. □

Let Π_6 be the predicate over $\mathcal{C}$ such that $\gamma \in \mathcal{C}$ satisfies Π_6 iff the distributed data structure maintained by the variables of Algorithm 10-11 forms a Proper Greatest Common Prefix Tree. We want $\forall p, \forall q_1, q_2 \in C_p, l_p = \text{GCP}(l_{q_1}, l_{q_2})$. Starting from Lemmas 6, 9, 10, 13, 17, 18, and 19, it remains to eliminate problematic cases expressed by conditions of Line 12.17 and Line 12.20. By the repeated executions of Lines 12.17-12.26, we can claim:

Theorem 4. *The system is self-stabilizing with respect to* Π_6.

5.3.4 Simulation Results

To capture what we can expect in terms of scalability, we simulated the protocol. In particular, we investigated the convergence time and the number of messages exchanged both w.r.t. the number of nodes.

The simulator is written as a Python script. The script randomly creates an initial faulty configuration of the network. By *randomly*, we mean that each node is created independently from the others. To create one node, it picks a randomly created label (on the Latin alphabet) of size between 1 and 20. It also chooses some nodes randomly from the set of already created nodes, to become the parent and the children of the node currently created. Thus, the initial graph is inconsistent

— prefix relationship may be wrong (*e.g.,* a node label may be prefixed by the label of its child), and the information about the neighbors may be incorrect. For example, p may consider its parent label is l although q is labeled $l' \neq l$, or p may assume q as its parent while p does not consider q is its child. We created the tree totally randomly to test the power of the proposed self-stabilizing protocol.

The protocol is launched at each node of the graph. We assume a discrete time. Each period of time is a *processing sample*. In other words, we decided that one period would begin when the first node starts the execution of the periodic rule, and would end when every node has triggered the periodic rule once and only once, and the set of actions resulting from it, *i.e.,* sending messages, processing messages, updating variables on any pertained node; have been executed. As we detailed in the proof, this set is finite, since the maximum set of messages generated by one execution of the periodic rule is finite. To implement the discrete sampling, processes are *synchronized*. But, the discrete time reflects the slowest processor rate. In other words, the scheduler simulated is fair.

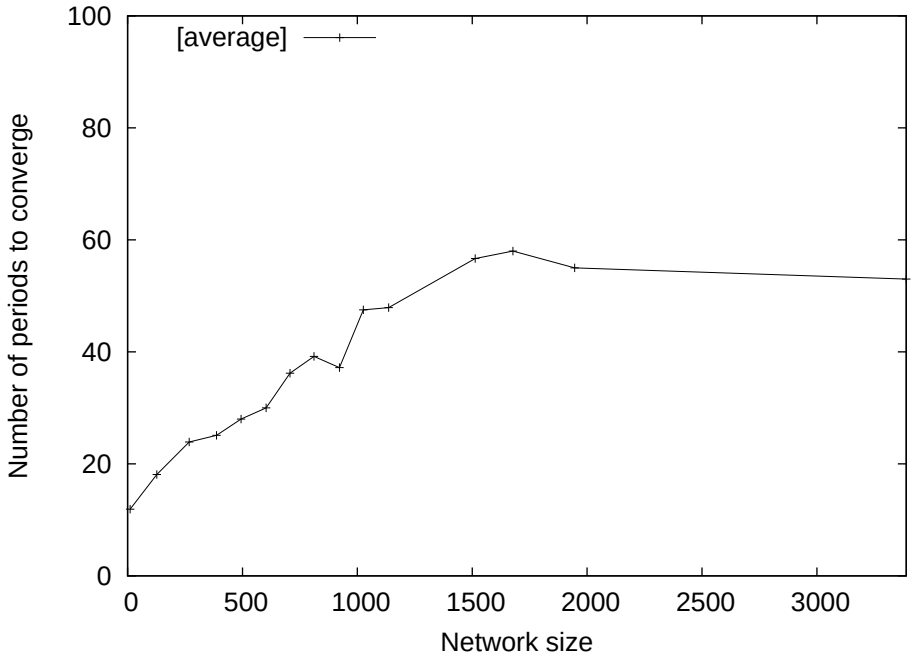


Figure 5.6: Message passing protocol simulation: convergence time.

Figure 5.6 shows the number of periods required in the average to converge, in function of the number of the final number of nodes in the tree. Recall that this number is equal to the initial number of distinct labels in the graph plus the number of labels created for the validity of the tree. The curve on Figure 5.6 shows that the number of periods required to converge increases very slowly when the size of the tree ranges from a couple of nodes to more than 3000. This suggests the convergence time grows at worst linearly, and with a very low slope (approximately 1/50). Figure 5.7 gives an average estimation for the number of messages each node exchanges during one period in function of the final number of nodes. Again, the curve suggests at worst a linear behavior. These two results show that, when the tree grows, both the amount of processing and the number of messages exchanged by the nodes (and thus the utilization of CPU and network resources) grows slowly, what indicates the

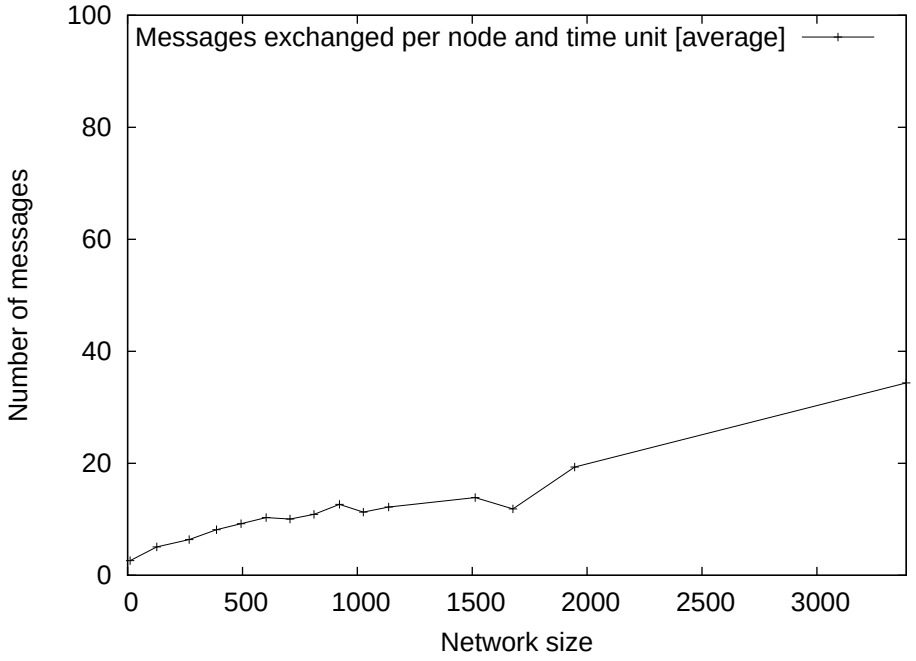


Figure 5.7: Self-stabilizing message passing protocol simulation: amount of messages.

scalability of the protocol.

Finally, we simulated clients' requests, *i.e.,* discovery requests looking up for services. In a regular use, discovery requests on a given service (or label) are encapsulated in a message sent to a randomly picked node. Then the message is routed until reaching the node labeled by the requested service.

We investigated if a prefix tree overlay enhanced with our self-stabilizing protocol, independently from its convergence time, allows the system to guarantee a certain level of availability. To this end, we simulated a prefix tree continuously undergoing failures, in a faster rate than the convergence time, under the same discrete-time conditions than for the previous experiments. On Figure 5.8, the X-axis expresses the number of nodes undergoing failures in percentage (0-10) of the total number nodes of the tree (about 500 in this experiment), at each period. The Y-axis gives the percentage of client's requests satisfied. A request is said to be satisfied if it reached its destination in the tree starting from a random entry node. The curve shows that this number is greatly improved when the self-stabilizing algorithm is performed — approximately from 5% to 40% and in spite of very bad conditions, *i.e.,* 10% of nodes failing at each period. The *basic* tree includes no other fault-tolerance mechanism, like replication.

Self-stabilization for tree-structured peer-to-peer systems. This last section presented a practical self-stabilizing protocol for the maintenance of a tree-structured P2P indexing system. While previous similar works mainly rely on theoretical coarse grain models, this protocol simply relies on message passing and is easily implemented on P2P platforms. We provided a comprehensive and formal correctness proof of the proposed protocol. We demonstrated that the convergence time and the amount of communications produced by the protocol increases slowly when the tree grows, indicating a good scalability. Moreover, we showed the ability of the protocol to greatly improve

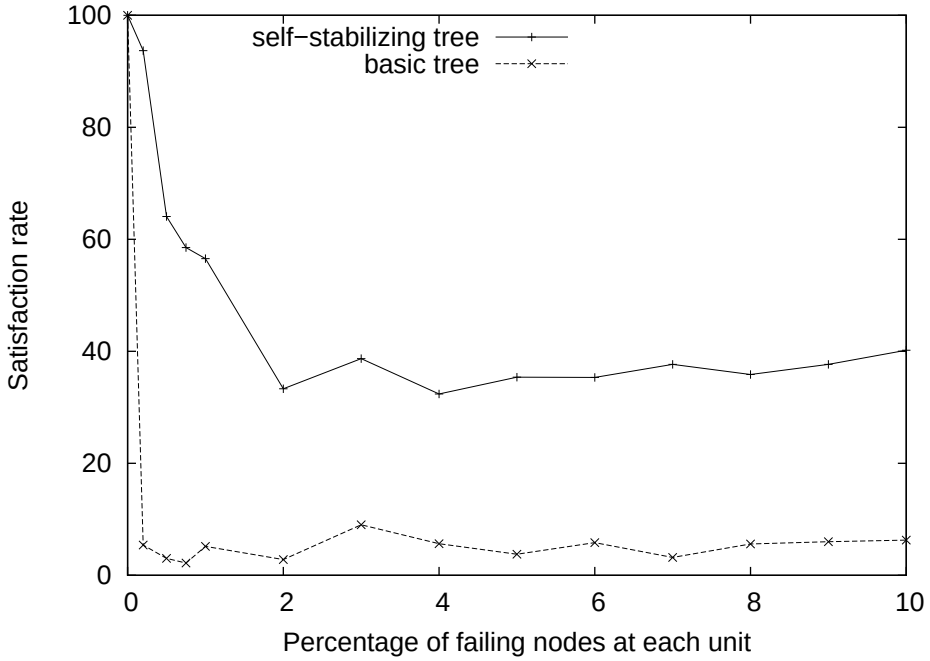


Figure 5.8: Simulation of the protocol: satisfaction rate of clients' requests.

fault tolerance in such architectures and thus increase the availability of service discovery systems.

5.4 Conclusion

In this chapter, we have presented three protocols proposing *best-effort* alternatives to the replication in prefix-tree peer-to-peer systems. The fault-tolerance provided by these algorithms outperforms replication algorithms in the sense that they allow to have an available consistent service discovery system after crashes, failures or wrong initializations of some variables of the nodes, in a finite time. Our first protocol is a message-passing algorithm allowing to reconnect and reorder disconnected subtrees in a system which underwent some peer crashes. The two following protocols are self-stabilizing but offer different and complementary contributions. The first protocol is written in a restricted model and assumes the logical topology to always be a rooted connected tree, but have the nice property to be *snap-stabilizing*, *i.e.,* is optimal in terms of stabilization time. The last protocol addresses the practical side of the problem, is written in a peer-to-peer oriented model, and is ready to be implemented on some actual platforms. A comprehensive correctness proof of the protocol is provided, but also simulation results highlighting the scalability of this protocol.

Algorithm 6 Reconnection Protocol for each node p

1.01 **Variables:** l_p, the label of p
f_p, the parent of p
l_{f_p}, the label of the parent of p
tf_p, temporary father of p
C_p, set of pairs $(identifier, label)$ of children of p
T_p, set of pairs $(identifier, label)$ of temporary children of p

2.01 **upon** receipt **of** <DISCONNECTED> **do**
2.02 $P :=$ Set of peers in the DHT (collected by a DHT traversal)
2.03 $N :=$ Set of logical nodes run on peers in P (collected by scanning the nodes in P)
2.04 $L :=$ Set of logical nodes in my subtree (collected using a PIF wave) using $C_p \cup T_p$

3.01 **if** $tf_p \neq \perp$ **then**
3.02 **send** <UNLINK, (p, l_p)> **to** tf_p
3.03 **if** $N \setminus L = \emptyset$ **then** //I am the root
3.04 $f_p := \perp$; $tf_p := \perp$
3.05 **else**
3.06 $tf_p :=$ random choice among $N \setminus L$
3.07 **send-sync** <LINK, (p, l_p)> **to** tf_p
3.08 **send** <HELLO, p> **to** tf_p

4.01 **upon** receipt **of** <HELLO, $list$> **from** q **do**
4.02 **if** $First(list) = p$ **then** //A cycle is detected
4.03 $leader := LeaderElection(list)$
4.04 **if** $p = leader$ **then**
4.05 Executes "**upon** receipt **of** <DISCONNECTED> **do**",
4.06 **elseif** $f_p \neq \perp$ **then**
4.07 **send** <HELLO, $list$> **to** f_p
4.08 **elseif** $tf_p \neq \perp$ **then**
4.09 $list := list + p$
4.10 **send** <HELLO, $list$> **to** tf_p
4.11 **else** // I am a false root still not linked or the real root
4.12 Executes "**upon** receipt **of** <DISCONNECTED> **from** ' **do**'
4.13 **if** $tf_p \neq \perp$ **then**
4.14 $list := list + p$
4.15 **send** <HELLO, $list$> **to** tf_p
4.16 **else** // I am the real root, so there is no cycle.
4.17 **send** <NOCYCLE> **to** $First(list)$

5.01 **upon** receipt **of** <NOCYCLE> **from** q **do**
5.02 **send** <MOVE, (p, l_p)> **to** tf_p
5.03 **send-sync** <UNLINK, (p, l_p)> **to** tf_p
5.04 $tf_p := \perp$

6.01 **upon** receipt **of** <LINK, (q, l_q)> **from** q **do**
6.02 $T_p := T_p \cup \{(q, l_q)\}$

7.01 **upon** receipt **of** <UNLINK, (q, l_q)> **from** q **do**
7.02 $T_p := T_p \setminus \{(q, l_q)\}$

Algorithm 7 Reorganization Protocol for each node p

8.01 **Variables:** l_p, the label of p
f_p, the parent of p
l_{f_p}, the label of the parent of p
C_p, set of pairs $(identifier, label)$ of children of p

9.01 **upon** receipt **of** <MOVE, fs> **from** q **do**
9.02 **if** $l_{fs} = l_p$ **then** // I send to myself that a fusion is needed.
9.03 **send** <MERGE, fs> **to** p
9.04 **elseif** $l_p \in$PREFIXES(l_{fs}) **then**
9.05 **if** $\exists s \in C_p | \, l_s \in$ PREFIXES(l_{fs}) **then**
9.06 // fs is in the subtree of s, Case (a) in Figure 5.1
9.07 **send** <MOVE,fs> **to** s
9.08 **elseif** $\exists s \in C_p | \, l_{fs} \in$ PREFIXES(l_s) **then**
9.09 // s is in the subtree of fs, Case (b) in Figure 5.1
9.10 $C_p := C_p \cup \{fs\} \setminus \{s\}$
9.11 **send** <MOVE,s> **to** fs
9.12 **elseif** $\exists s \in C_p \mid |l_p| < |PGCP(l_s, l_{fs})|$ **then**
9.13 // fs and s have a PGCP which is greater than l_p
9.14 // Case (c) in Figure 5.1
9.15 $Newnode(PGCP(l_{fs}, l_s), s, fs)$; $C_p := C_p \setminus \{s\}$
9.16 **else** // fs is one of my children, Case (d) in Figure 5.1
9.17 $C_p := C_p \cup \{fs\}$
9.18 **else**
9.19 **if** $f_p \neq \perp$ **then**
9.20 **send** <MOVE,fs> **to** f_p
9.21 **else**
9.22 **if** $l_{fs} \in$ PREFIXES(l_p) **then**
9.23 // I am in the subtree of fs
9.24 **send** <MOVE,p> **to** fs
9.25 **else** // p and fs are siblings
9.26 $C_p := C_p \cup Newnode(PGCP(l_{fs}, l_f), fs, p)$

10.01 **upon** receipt **of** <MERGE,fs> **from** q **do**
10.02 GLUING(q)
10.03 Sorting of C_p in the lexicographic order in Table t_s
10.04 **for** $i = 0$ **do** $t_s.length() - 2$
10.05 **if** $l_{t_s[i]} = l_{t_s[i+1]}$ **then**
10.06 **send** <MERGE, $t_s[i+1]$> **to** $t_s[i]$
10.07 $i := i + 1$
10.08 **elseif** $l_{t_s[i]} \in$PREFIXES $(l_{t_s[i+1]})$ **then**
10.09 **send** <MOVE,$t_s[i+1]$> **to** $t_s[i]$
10.10 $C_p := C_p \setminus \{t_s[i+1]\}$
10.11 $i := i + 1$
10.12 **elseif** $l_p < PGCP(l_{t_s[i]}, l_{t_s[i+1]})$ **then**
10.13 $C_p := C_p \cup Newnode(PGCP(l_{t_s[i]}, l_{t_s[i+1]}), t_s[i], t_s[i+1])$
10.14 $C_p := C_p \setminus \{t_s[i], t_s[i+1]\}$
10.15 $i := i + 1$

Algorithm 8 Snap-Stabilizing PGCP Tree — Variables, Macros, and Actions.

Variables: l_p, the label of p
$C_p = \{c_1, \ldots, c_k\}$
$S_p = \{I, B\}$ if p is the `root`, $\{I, H\}$ if p is a leaf node, $\{I, B, H\}$ otherwise

Macros: $f_p \equiv \{q : \ p \in C_q\}$
$SameLabel_p(L) \equiv \{c \in C_p : \ (l_c = L)\}$
$SameGCP_p(L) \equiv \{c_1, c_2, \ldots, c_k \in C_p : \ GCP(c_1, c_2, \ldots, c_k) = L\}$
$SamePGCP_p(L) \equiv SameGCP_p(L) \setminus \{c \in SameGCP_p(L) : \ l_c = L\}$

Actions:

			{**For the `root` node**}	
$InitBroadcast$	::	$S_p = I \wedge (\forall c \in C_p : \ S_c = I)$	$\longrightarrow$	$S_p := B$;
$InitRepair$	::	$S_p = B \wedge (\forall c \in C_p : \ S_c = H)$	$\longrightarrow$	**HEAPIFY**();**REPAIR**(); $S_p := I$;
			{**For the internal nodes**}	
$ForwardBroadcast$	::	$S_p = I \wedge S_{f_p} = B \wedge (\forall c \in C_p : \ S_c = I)$	$\longrightarrow$	$S_p := B$;
$BackwardHeap$	::	$S_p = B \wedge S_{f_p} = B \wedge (\forall c \in C_p : \ S_c = H)$	$\longrightarrow$	**HEAPIFY**(); $S_p := H$;
$ForwardRepair$	::	$S_p = H \wedge S_{f_p} = I \wedge (\forall c \in C_p : \ S_c \in \{H, I\})$	$\longrightarrow$	**REPAIR**(); $S_p := I$;
$ErrorCorrection$	::	$S_p = B \wedge S_{f_p} \in \{H, I\}$	$\longrightarrow$	$S_p := I$;
			{**For the leaf nodes**}	
$InitHeap$	::	$S_p = I \wedge S_{f_p} = B$	$\longrightarrow$	$S_p := H$
$EndRepair$	::	$S_p = H \wedge S_{f_p} = I$	$\longrightarrow$	$S_p := I$;

Algorithm 9 Snap-Stabilizing PGCP Tree — Procedures.

```
10.01 Procedure HEAPIFY()
10.02     if l_p ≠ PGCP ({l_c | c ∈ C_p}) then
10.03         C_p := C_p ∪ {NEWNODE (l_p, H, {})}
10.04         l_p := GCP({l_c : c ∈ C_p})
10.05         for all c ∈ C_p : l_c = l_p do
10.06             C_p := C_p ∪ C_c \ {c}
10.07             DESTROY(c)
10.08     done

11.01 Procedure REPAIR()
11.02     while ∃(c_1, c_2) ∈ C_p : l_{c_1} = l_{c_2} do
11.03         C_p := C_p ∪ {NEWNODE(l_{c_1}, H, C_{s: s∈SameLabel(l_{c_1})})}
11.04         for all c ∈ SameLabel_p(l_{c_1}) do
11.05             DESTROY(c)
11.06         done
11.07     done
11.08     while ∃c ∈ C_p : SamePGCP_p(l_c) ≠ ∅ do
11.09         C_p := C_p ∪ {NEWNODE(l_c, H, C_c ∪ SamePGCP_p(l_c)}
11.10         C_p := C_p \ SamePGCP_p(l_c)
11.11         DESTROY(c)
11.12     done
11.13     while ∃(c_1, c_2) ∈ C_p : |GCP(l_{c_1}, l_{c_2})| > |l_p| do
11.14         C_p := C_p ∪ {NEWNODE(GCP(l_{c_1}, l_{c_2}), H, SameGCP_p(GCP(l_{c_1}, l_{c_2}))}
11.15         C_p := C_p \ SameGCP_p(GCP(l_{c_1}, l_{c_2}))
11.16     done
```

Algorithm 10 Periodic rule, on process p

11.01 **Variables:** $\widehat{p} = (p, l_p)$, id and label of p
$\widehat{f_p} = (f_p, l_{f_p})$, id and label of the parent of p
$\widehat{C_p} = \{\widehat{q_1} = (q_1, l_{q_1}), \ldots, \widehat{q_k} = (q_k, l_{q_k})\}$, a finite set of pairs (ids,labels), children of p
$T_q, \forall q \in C_p$ time before considering q as not its child anymore
11.02 **Macros:** $C_p \equiv \{q \mid (q, l_q) \in \widehat{C_p}\}$, set of totally ordered ids of children of p

12.01 **Upon** `TimeOut` **do**
12.02 **if** $f_p = p$ **then** $f_p := \perp$
12.03 **if** $p \in C_p$ **then** $\widehat{C_p} := \widehat{C_p} \setminus \{\widehat{p}\}$
12.04 **if** $f_p = \perp$ **then**
12.05 **if** $l_p = \epsilon$ **then**
12.06 $q :=$ GetEpsilon()
12.07 **if** $q < p$ **then**
12.08 $\widehat{C_p} := \widehat{C_p} \cup \{(q, \epsilon)\}$
12.09 **send**(<UpdateParent, $\widehat{p}$>, q)
12.10 **else**
12.11 $new :=$ GetNewProcess()
12.12 **send**(<Host, $(\epsilon, \perp, \{\widehat{p}\})$>, new)
12.13 $\widehat{f_p} := (new, \epsilon)$
12.14
12.15 **while** $\exists q \in C_p \mid l_q = l_p$ **do**
12.16 **send**(<Merge, $\widehat{p}$>, q)
12.17 **while** $\exists (q_1, q_2) \in C_p^2 \ : \ l_p \in$ Prefixes$(l_{q_1}) \wedge l_{q_1} \in$ Prefixes$(l_{q_2}))$ **do**
12.18 **send**(<UpdateParent, $\widehat{q_1}$>, q_2)
12.19 $\widehat{C_p} := \widehat{C_p} \setminus \{\widehat{q_2}\}$
12.20 **while** $\exists (q_1, q_2) \in C_p^2 \ : \ l_p \in$ Prefixes$(l_{q_1}) \wedge l_p \in$ Prefixes$(l_{q_2}) \wedge |GCP(l_{q_1}, l_{q_2})| > |l_p|$ **do**
12.21 $l_{new} :=$ GCP(l_{q_1}, l_{q_2})
12.22 $new :=$ GetNewProcess()
12.23 **send**(<Host, $(l_{new}, p, \{q_1, q_2\})$>, new)
12.24 **send**(<UpdateParent, $\widehat{new}$>, q_1)
12.25 **send**(<UpdateParent, $\widehat{new}$>, q_2)
12.26 $\widehat{C_p} := \widehat{C_p} \setminus \{\widehat{q_1}, \widehat{q_2}\} \cup \{\widehat{new}\}$
12.27 **if** $f_p \neq \perp$ **then**
12.28 **send**(<Parent?, $\widehat{p}$>, f_p)

Algorithm 11 *Upon receipt* rules, on process p

13.01 **upon** receipt **of** <PARENT?, $\widehat{q}$> **do**
13.02 **if** $l_p \in$ PREFIXES$(l_q) \wedge$ **send**(<CHILD, $\widehat{p}$>, q) **then**
13.03 $\widehat{C_p} := \widehat{C_p} \cup \{\widehat{q}\}$
13.04 **else**
13.05 **send**(<ORPHAN, $\widehat{p}$>, q)
13.06 $\widehat{C_p} := \widehat{C_p} \setminus \{\widehat{q}\}$

14.01 **upon** receipt **of** <CHILD, $\widehat{q}$> **do**
14.02 **if** $f_p = q$ **then**
14.03 $l_{f_p} := l_q$

15.01 **upon** receipt **of** <ORPHAN, $\widehat{q}$> **do**
15.02 **if** $f_p = q$ **then**
15.03 $f_p := \perp$

16.01 **upon** receipt **of** <UPDATEPARENT, $\widehat{q}$> **do**
16.02 **if** $(l_q \in$ PREFIXES$(l_p)) \wedge$ **send**(<PARENT?, $\widehat{p}$>, q) **then**
16.03 $\widehat{f_p} := \widehat{q}$

17.01 **upon** receipt **of** <HOST, $l, f, \widehat{C}$> **do**
17.02 STARTPROCESS $(l, f, \widehat{C})$

18.01 **upon** receipt **of** <MERGE, $\widehat{q}$> **do**
18.02 **if** $(f_p = q) \wedge (l_q \in$ PREFIXES$(l_p))$ **then**
18.03 $\forall q' \in C_p$, **send**(<GRANDPARENT, $\widehat{f_p}$, $\widehat{q}$>, q')

19.01 **upon** receipt **of** <GRANDPARENT, $\widehat{newf}$, $\widehat{q}$> **do**
19.02 **if** $(f_p = q) \wedge (l_q \in$ PREFIXES$(l_p))$ **then**
19.03 $\widehat{f_p} := \widehat{newf}$
19.04 **send**(<GFDONE, $\widehat{p}$>, q)

20.01 **upon** receipt **of** <GFDONE, $\widehat{q}$> **do**
20.02 **if** $(q \in C_p) \wedge (l_p \in$ PREFIXES$(l_q))$ **then**
20.03 $\widehat{C_p} := \widehat{C_p} \setminus \{\widehat{q}\}$
20.04 **if** $\widehat{C_p} = \emptyset$ **then**
20.05 **send**(<MDONE, $\widehat{p}$>, f_p)
20.06 KILL(p)

21.01 **upon** receipt **of** <MDONE, $\widehat{q}$> **do**
21.02 **if** $(q \in C_p) \wedge (l_p \in$ PREFIXES$(l_q))$ **then**
21.03 $\widehat{C_p} := \widehat{C_p} \setminus \{\widehat{q}\}$

Chapter 6

Prototype Implementation and Applications

The purpose of this last chapter[1] is threefold:

1. Our first point is to present our early development works around the service discovery problem. This preliminary work has focused on improving the scalability of one particular grid middleware following the GridRPC standard [137], and whose one of the main features is service discovery, called DIET [40]. Our architecture extends DIET by connecting its main service discovery components in a peer-to-peer network in which the propagation of requests has been implemented following several approaches and calls upon the JXTA toolbox [145]. Experimental results show the viability and scalability of this extension.

2. Our second concern is to better capture the viability that we can expect of our architecture, in terms of deployment over a real platform. Although the scalability, load-balancing, and fault-tolerance has been showed by analysis and simulation, the concepts must now be confronted to real settings. In a second section, we present our prototype, based on the JXTA toolbox [145], that implements the design of our architecture described in Chapter 3. Early experimentation results are given.

3. Our third concern is to give an insight in the use of our concept to solve a particular problem related to large scale resource discovery. We present the use of our concepts for the control of a network resources reservation service. More generally, our application is the reservation of resources with *network-awareness*, *i.e.,* be able to make a relevant selection according to the network characteristics, for instance if the amount of data to be transferred between the client and the server is large. We discuss the motivation for this problem, the adaptation, combination of our algorithms and structures and the results of our first deployments on the Grid'5000 platform [33], a nation-wide infrastructure gathering about 5000 CPUs dedicated to research purposes.

[1] Part of the work presented in this chapter has been published in [CDPT05], the second and third sections are the result of a collaboration with the RESO team from INRIA (http://www.ens-lyon.fr/LIP/RESO/)

6.1 A Peer-to-Peer Extension of Network-Enabled Servers.

In this first section, we present our first software contribution, a preliminary work conducted from 2004 to 2005 whose goal was to identify and break the limitations of the service discovery in a grid middleware. This architecture relies on the DIET middleware [40], propagation algorithms and the JXTA toolbox.

6.1.1 The GridRPC Model

Among existing grid middleware approaches, one simple, powerful, and flexible approach consists in using servers available in different administrative domains through the classical Client/Server or Remote Procedure Call (RPC) paradigm. *Network Enabled Servers* environments implement this model, also called *GridRPC*. Clients submit computation requests to a scheduler whose goal is to find a server available on the grid. In the next section, we focus on the GridRPC paradigm.

Remote Procedure Call. The Remote Procedure Call (RPC) paradigm is one of the most used in the programmers' community. In its object-oriented version, methods are implemented by server objects that clients can access through an interface describing the set of methods the object provides. In a distributed environment, *i.e.,* where objects are distant, the invocation of a method by a client triggers the sending of a request to the server object. However, the set of communications generated is hidden to the user inside simple method invocations. To this end, the client has an object — the *stub*, representing the server and the server is encapsulated in an object in charged of the reception of communications, called the *skeleton*. Many implementations of RPC exists. Among the most known are Java RMI [84] and the CORBA standard [133].

Network-Enabled Servers RPC. Since the end of the nineties, the RPC paradigm has appeared to be a good candidate to build Problem Solving Environments (PSE) for numerical applications on the Grid. Built upon the RPC paradigm, the *Network-Enabled Servers* model [104] basically aims at providing access to computational facilities via potential servers to end users wishing to solve their problems that cannot be solved using a single machine and require multiple potential servers for request computation, so as to find the result in appropriate time. They usually have five different components. **Clients** that submit problems they have to solve to **Servers**, a **Database** that contains information about software and hardware resources, a **Scheduler** that chooses an appropriate server depending on the problem sent and the information contained in the database, and finally **Monitors** that acquire information about the status of the computational resources.

A GridRPC API. Under the umbrella of the Open Grid Forum [5], the early NES model gave birth to the *GridRPC* paradigm, a grid-enabled version of the RPC, which intends to ease the programming of applications over the grids for programmers already familiar with the RPC paradigm. The GridRPC working group of the Open Grid Forum intends in particular to define a standard GridRPC API. This API aims to offer to non-grid specialist users a high-level API to *gridify* their application. According to the guide for testing the interoperability of a given middleware with the GridRPC API specifications [137], current GridRPC compliant middleware are Ninf-G, GridSolve, and Diet.

GridRPC Middleware. Ninf-G [136][2] provides the GridRPC API on top of the Globus Toolkit, which, as previously noted, provides a reference implementation of standard protocols and APIs for Grid computing. Ninf-G is maintained by the Ninf project in National Institute of Advanced Industrial Science and Technology (AIST), and several universities in Japan. GridSolve [156][3] is a Client/Server system which provides remote access to computational resource, both hardware and software. It is built upon standard Internet protocols, like TCP/IP sockets. GridSolve provides the GridRPC API as C interface, and another API as Matlab interface. GridSolve is maintained by the University of Tennessee Knoxville in the United States. In GridSolve, the processing of requests submitted by clients and the scheduling of jobs is achieved by a central device, which is a bottleneck and a single point of failure of the system. Having this in mind, The Distributed Interactive Engineering Toolbox (DIET) project [40] originally focused on the scalability issue in GridRPC environments. In the following section, we briefly review DIET features, describe its architecture and elements and highlight its scalability limits.

6.1.2 DIET

The DIET middleware is focused on the development of scalable middleware by distributing the scheduling task across multiple agents. DIET consists of a set of elements that can be used together to build applications using the GridRPC paradigm. DIET is able to find an appropriate server according to the information given in the client's request (problem to be solved, size of the data involved), the performance of the target platform (server load, available memory, communication performance) and the local availability of data stored during previous computations. The scheduler is distributed in a hierarchy of scheduling agents. In order to avoid unnecessary communication when dependences exist between different requests, DIET data management allows persistent data to stay within the system for future re-use.

DIET Architecture

The DIET architecture follows a hierarchical approach and is based upon several elements. First a **Client** is an application that uses DIET to solve problems in a RPC mode. Different kinds of clients should be able to connect to DIET from a web page, a Problem Solving Environment such as Scilab, or from a program written in C/C++, Fortran or Java. Traditionally a centralized device in other NES systems such as NetSolve or Ninf, the DIET scheduler is scattered across a hierarchy of *Agents*. Figure 6.1 shows such a hierarchy.

Scheduling Agents

A **Master Agent** (MA) is the entry point of the DIET environment and thus receives computation requests from clients. These requests refer to some problems that can be solved by registered servers. These problems can be listed on a reference web page. A client can be connected to a MA by a specific name server or a web page which stores the various MA locations. Then the MA collects computation abilities from the servers and chooses the best one according to some scheduling heuristics (deadline scheduling, shortest completion time first, maximization of the requests throughput, ...). A reference to the server chosen is sent back to the client.

[2] http://ninf.apgrid.org/
[3] http://icl.cs.utk.edu/gridsolve/

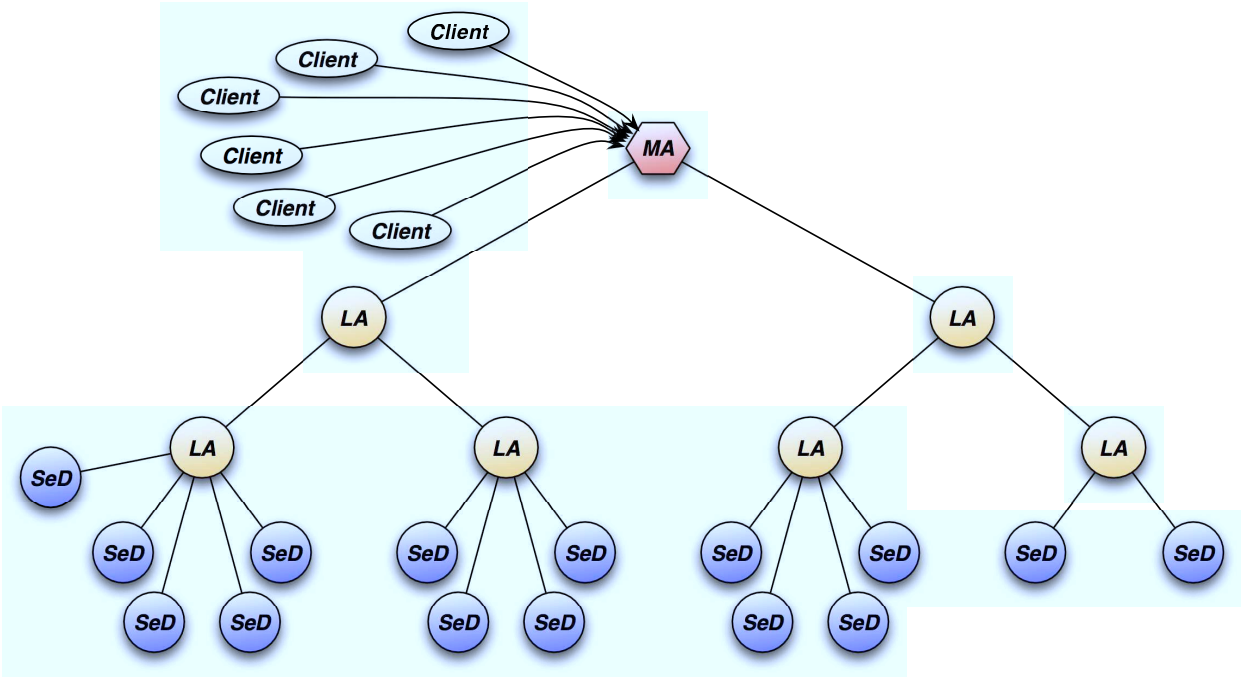


Figure 6.1: DIET hierarchical organization.

A Master Agent relies on a hierarchy of agents to gather information and scheduling decisions. An **Agent** aims at transmitting requests and information between MAs and LAs. A **Local Agent** (LA) aims at transmitting requests and information between Agents and several servers. The information stored on an Agent is the list of servers registered on Local Agents of its subtree, the problems they are able to solve and information about the data distributed in this subtree. Depending on the underlying network topology, Agents may be deployed between the MA and the LAs. The scheduling and the gathering of information is thus distributed in the tree.

Server Daemons

Computations are done by servers (both sequential and parallel) in front of which we have **Server Daemons (SeD)**. A SeD encapsulates a computational server, typically on the entry point of a parallel supercomputer or cluster. The information stored on a SeD is a list of data available on its server (with their distribution and the way to access them), the list of problems that can be solved on it, and all information concerning its load (memory and/or number of resources available, ...). A SeD registers to a Local Agent that becomes its parent and declares the problems it can solve to it. A SeD can give performance predictions for a given problem using the performance evaluation module (CoRI).

DIET Scalability Limits

We can identify three drawbacks of DIET related to scalability.

1. **Static Nature.** Such static hierarchies do not cope with the dynamicity of nodes at large scale, resulting in difficulties to deploy such hierarchies on large grids. As a consequence, most of those hierarchies are not deployed among more than one administrative domain. Moreover, the clients are given an entry point statically. One need for the client is to dynamically choose its *best* MA considering metrics such as latency.
2. **Bottleneck.** The hierarchy has a unique entry point (the MA) for every clients. This involves a probability of creating a bottleneck growing with the number of requests submitted by clients.
3. **Unreachable Services.** Real life use cases show that services are quite often deployed among only one hierarchy for many reasons (for instance data locality or security). One key purpose of computational grids is to make services available for clients anywhere in wide area networks. So there is a strong need for making services available for clients, who does not necessarily know the entry point of the hierarchy providing the requested service.

6.1.3 DIET$_J$: A P2P extension of DIET

In this section, we present DIET$_J$, a peer-to-peer extension of DIET addressing the different scalability issues previously mentioned. After a presentation of the DIET$_J$ architecture and its goals, we accurately describe the propagation of the clients' requests inside the platform, using several algorithm and discuss their properties. Finally, we give experimental results of this architecture conducted on several clusters connected by a high-speed network.

DIET$_J$ Overview

The aim of DIET$_J$ is to dynamically connect together geographically distributed DIET hierarchies to gather services on-demand and improve the scalability of service discovery. This new architecture addresses the drawbacks described at the end of the previous section and have the following properties:

1. **Dynamic Connection of Hierarchies.** To increase the scalability of DIET over the grid, we dynamically build a *multi-hierarchy* by connecting the entry points of the hierarchies (Master Agents) together. The multi-hierarchy is built on-demand by a Master Agent after it failed to find the service requested by a client inside its own hierarchy. The clients have now the ability to discover and connect the MA with the best latency/locality according to them.
2. **Distribution of the MA Load.** The entry point for each client being dynamically chosen, the bottleneck on the unique Master Agent is avoided. Master Agents are connected in an unstructured Peer-to-Peer fashion (without any maintenance of the neighborhood/ routing information.)
3. **An Access to Remote Domain Services.** Whereas DIET hierarchies were unable to communicate together, services are here gathered between hierarchies thus providing to clients a front door to resources put in common at large scale in a transparent way.

Our extension is based on JXTA [145]. JXTA defines a set of protocols for building Peer-to-Peer (P2P) applications on top of the physical network. The basic logic entity of the JXTA virtual network is the peer. Each JXTA entity (peers, pipes, services) is uniquely identified by an **advertisement**. JXTA provides dynamic mechanisms of discovery of JXTA entities, thus allowing any peer to dynamically address any other peer on the network.

DIET$_J$ Architecture

The DIET$_J$ architecture, shown in Figure 6.2, connects several DIET hierarchies by their root (*i.e.*, Master Agents), using JXTA communication protocols.

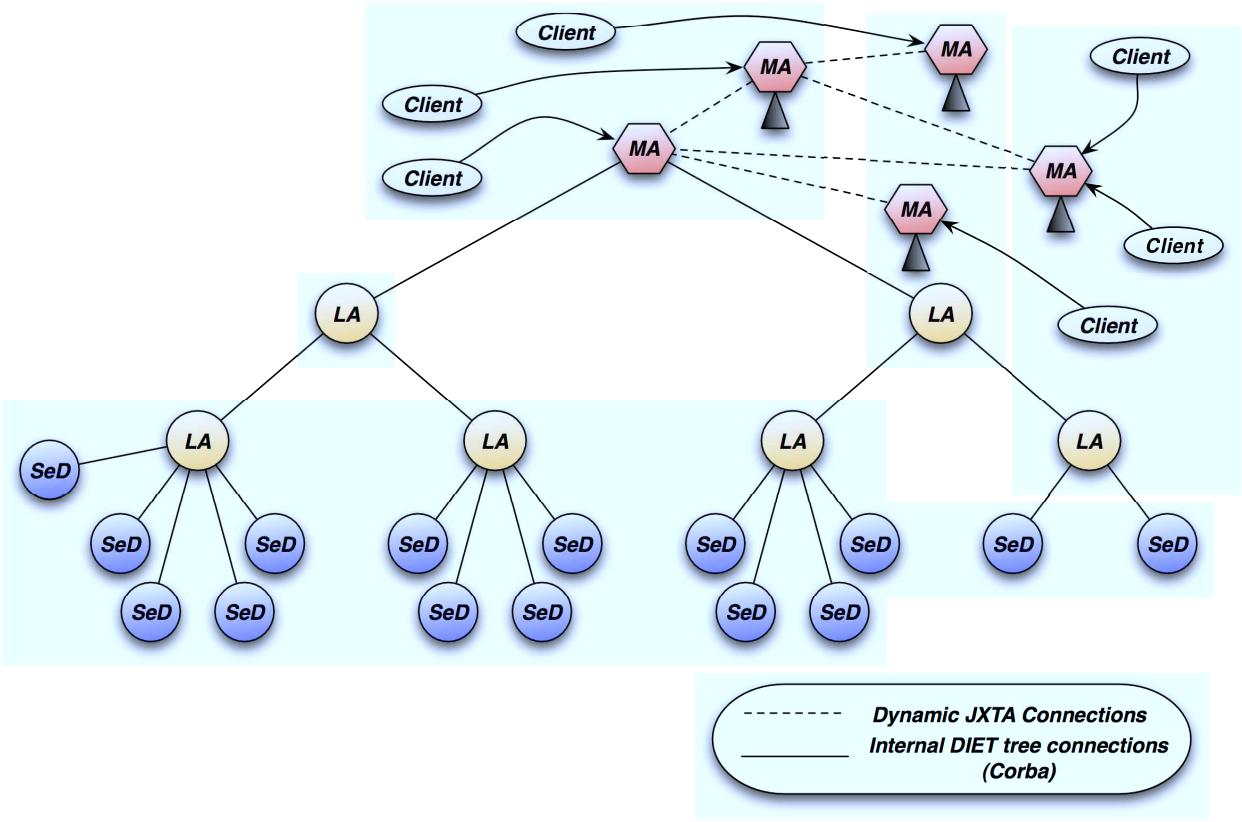


Figure 6.2: DIET$_J$ architecture.

As illustrated on Figure 6.3, the Master Agent's internal architecture is divided into three parts.

- **The JXTA Part.** The JXTA part of the Master Agent is a peer on the JXTA virtual network. This is the connection point of this Master Agent to other ones. This part is a java bytecode.
- **The DIET Part.** The DIET part is the traditional DIET Master Agent, root of a DIET hierarchy of Agents and Local Agents, allowing the discovery of servers that registered to this hierarchy. This part is a C program relying on libraries generated from the DIET C code.
- **The Interface.** To cooperate, Java (JXTA native language) and C (DIET native language) need an interface. We use the JNI technology [85] allowing to call C functions from a Java program, and the data conversion between the two environments.

The Multi Master Agent System. The set of alive MAs that a first MA is able to discover using JXTA discovery mechanism and thus able to connect using some JXTA connection facilities can put

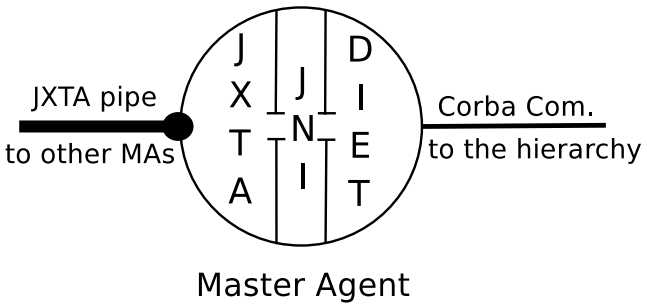


Figure 6.3: Master Agent Internal Architecture.

their resource in common to build a JXTA network of MAs, refer to as a *Multi-MA*, henceforth. As we mentioned before, a MA is a JXTA peer. As a consequence, it can be discovered by other peers on the JXTA network by an advertisement containing a particular name, common to all of them, *e.g.,* `DIET_MA`. In order for one MA to be discovered, this advertisement is published at the beginning of one MA's life and then republished periodically with a short lifetime to avoid other peers to try to bind a dead MA, and thus easily take into account the dynamic nature of the platform. At load time, the Java part of the MA launches the DIET part via JNI, and waits for requests. When receiving a client's request, the DIET part is called that submits the request to the hierarchy. If the submission to the DIET hierarchy retrieves no servers providing the requested service, the JXTA part starts the construction of a multi-MA. First, it launches a discovery process to have knowledge of other MAs in the network (thanks to their JXTA advertisements). Second, it propagates the request to them, triggering the search for the service within these hierarchies. When the JXTA part has received responses from all other MAs (or when a timeout is reached), the responses are merged and sent back to the client, who has not been aware that a multi-hierarchy has been temporarily built.

Dynamic Connections. Connections between the MAs are created when needed (if the service was not found). Dynamic connections between the MAs allow to transparently perform the service discovery in a dynamic multi-hierarchy, using JXTA advertisements. The communication between the agents inside one hierarchy are still static as we believe that small hierarchies are installed within each administrative domain. At this level, performance are not much variable, new elements are not frequently added, and the whole hierarchy will remain stable during its lifetime.

6.1.4 Traversing the DIET$_J$ multi-hierarchy

Discussion

We now discuss approaches and algorithms implemented for propagating the clients' requests in the JXTA network of Master Agents and gather information about servers of several hierarchies.

Discovering the MAs, then Discovering the Servers. It is important to note that the multi-hierarchy construction is divided into two parts. The first step is the **MA discovery** and aims at discovering MAs reachable on the network, thanks to the JXTA discovery process (we discuss this process in the next paragraph). Once a peer has been discovered, *i.e.,* if its advertisement (containing its name and information required to bind it, *e.g.,* an input pipe advertisement), the connection still

needs to be established. The second step is the **service discovery** and consists in exploring the multi-hierarchy composed of the MAs discovered in the first step, looking for the service requested within each DIET hierarchy.

JXTA Discovery Mechanisms. JXTA 2.x provides a hybrid mechanism combining DHT-like mechanisms and random walks [150] to achieve the discovery of advertisements, *e.g.,* advertisement named `DIET_MA`. Again, we choose not to use the hybrid DHT mechanism to avoid its maintenance overhead, its lack of exhaustiveness (when it fails to retrieve the advertisement by the hash function, it uses a less-efficient random walk method) and cope with the on-demand fashion of the multi-hierarchy designed. Thus, we use the JXTA discovery mechanism based on flooding among the peers. Once the MA's references obtained, an algorithm optimizing the traversal of the multi-hierarchy (the graph of MA connections) is used to connect MAs together and propagate the request through the multi-hierarchy.

Implementation

The propagation of the request in the graph of Master Agents has been implemented with two algorithms.

Propagation as an Asynchronous Star Graph Traversal. The propagation of a request has first been implemented as an intuitive asynchronous star graph traversal. This scheme is illustrated by Figure 6.4. One MA r found no server providing the service requested by a client in its own hierarchy — Figure 6.4 (a). It discovers other MAs with the JXTA discovery process and forwards the request asynchronously to all MAs previously discovered, using a simple JXTA multicast pipe instruction (b). On receipt of the forwarded request, each MA collects the servers able to solve the problem in its own hierarchy. On the figure, some finds it (in deep grey) and some do not find it (in light grey) (c). Once the DIET hierarchy interrogated, they send the response back to r (d) that collects and merges responses to create the final response message sent back to the client (e). Using this first algorithm, the propagation systematically builds a star graph, the MA initiating the propagation being the root of the star. This scheme is referred to as $STAR_{async}$ algorithm henceforth.

Propagation as an Asynchronous PIF Scheme. The request propagation has also been implemented following a modified version of the asynchronous *Propagation of Information with Feedback* (PIF) scheme, in order to obtain an unstructured and adaptive multi-hierarchy traversal. A complete description of the basic PIF algorithm can be found in [45, 128]. Our version is detailed by Algorithm 12. Figure 6.5 describes a scenario of propagation in a DIET multi-hierarchy, applying the two following phases:

1. The **Broadcast phase**: The MA that received the request, from the client (and is unable to find a server providing the requested service within its own hierarchy) initiates the wave — refer to Lines 1.05-1.09 and so is the root, denoted r. Similarly as done using the STAR_{async} algorithm, it forwards the request to the set of other MAs it has previously discovered (IDs in the set denoted $agentList$). r then waits for responses of MAs in $agentList$. Then, a MA m receiving a forwarded request checks if it has already received it. If not, the MA that sent the request becomes its parent (Lines 1.10-1.14). m immediately propagates the request on its turn to the MAs in $agentList$, except those that were on the path from r to m, contributing to

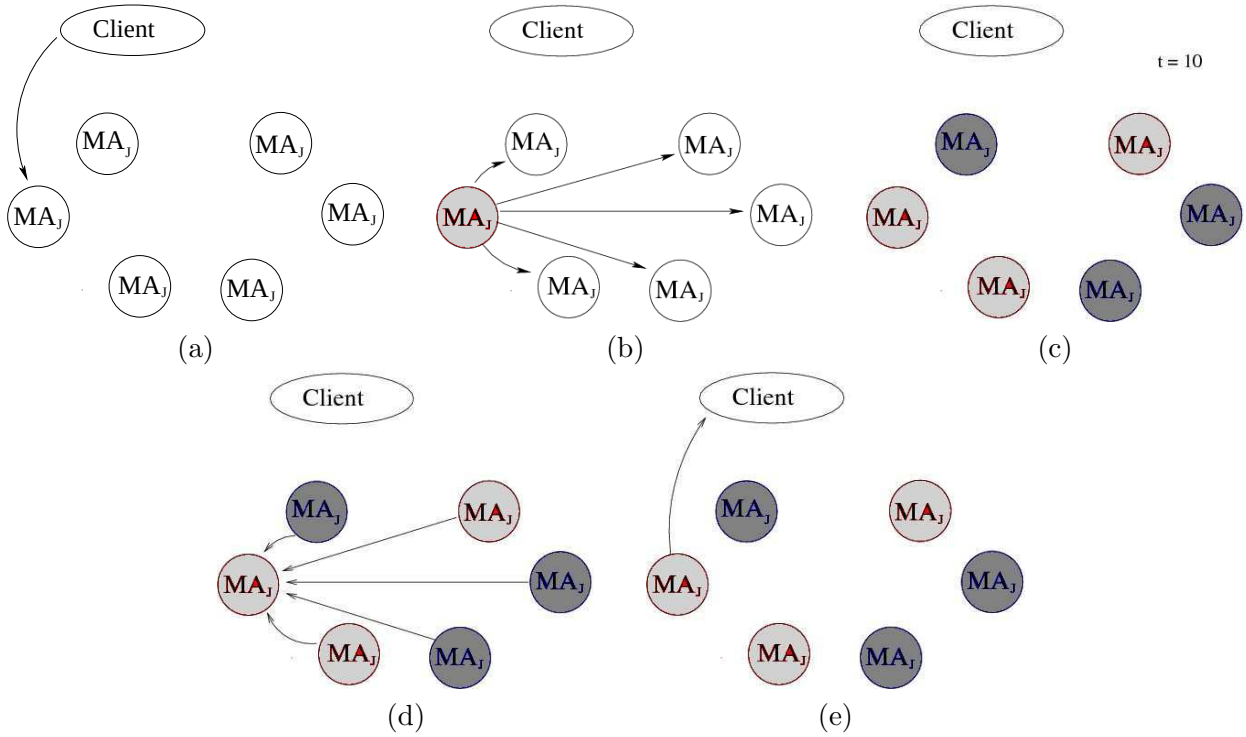


Figure 6.4: Scenario of a STAR$_{async}$ traversal.

build a time optimal spanning tree on the set of MA, rooted at r. m also calls its own DIET hierarchy to collect the servers able to solve the problem specified by the client and sends the DIET response to its parent.

2. The **Feedback phase**: r waits for the responses of MAs in *agentList* during a finite time using a timeout. Each MA in the tree built during the broadcast phase receives responses from nodes in their subtree containing sets of servers, denoted SL, and forwards them to their parent, along with the servers found in their own DIET hierarchy — refer to Lines 1.17-1.18.

Let us call this algorithm *PIF$_{async}$*. The PIF$_{async}$ algorithm builds an *on-demand optimal tree* for a given root for each request, thus balancing the load among the MAs graph as the number of requests increases and also avoiding overloaded links. It was shown in [128] that in asynchronous environments, the PIF scheme is the fastest possible to reach every network nodes, messages following the fastest links during the broadcast phase. In other words, the dynamic tree built during the propagation is time optimal at time of its creation, and the feedback phase follows the links of this optimal tree. The number of messages required is $O(n^2)$ in the worst case. Note that the STAR$_{async}$ algorithm also provides a particular PIF scheme, in which messages always follow the same links, ignoring their heterogeneity and communication load.

6.1.5 Experimenting DIET$_J$.

In this section, we discuss experimental results of the implementation of DIET$_J$ with the algorithms previously described.

Algorithm 12 Asynchronous PIF-based request propagation, on node (MA) p.

1.04 **Variables:** f_p: father of p
$servList$: list of servers satisfying req

1.05 **upon** receipt **of** $<req, agentList>$ **from** Application Layer **do**
1.06 $Father := \perp$
1.07 $servList := \emptyset$
1.08 **for** $q \in agentList$ **do**
1.09 **send** $<req, agentList \setminus \{p\}>$ **to** q

1.10 **upon** receipt **of** $<req, agentList>$ **from** q **do**
1.11 **if** $f_p \neq \perp$ **then**
1.12 $f_p := q$
1.13 **for** $q \in agentList$ **do**
1.14 **send** $<req, agentList \setminus \{p\}>$ **to** q
1.15 $servList :=$DietCall()
1.16 **send** $<req, servList>$ **to** f_p

1.17 **upon** receipt **of** $<req, SL>$ **from** q **do**
1.18 **send** $<req, SL>$ **to** f_p

Experimental Platform. Our experimental platform relies on several clusters connected through the 2.5 Gb/s VTHD network in France (see Figure 6.6). In 2005, part of the VTHD clusters have been integrated to the Grid'5000 platform. The clusters used are equipped with Intel quadri-processors Xeon 2.4 GHz and Intel bi-processors Xeon 2.8 GHz. One MA runs on each node and one client sends one or multiple requests to MAs. Based on previous experiments inside one unique hierarchy [42], where it was shown that a unique DIET Master Agent is able to have hundreds of servers in its hierarchy and remain efficient with a very high number of simultaneous requests, we here experimented connections of the MAs graph without underlying hierarchies.

Experiments with Homogeneous Network Performance. We started our experiments with a low and homogeneous traffic load, by varying the number of MAs in order to estimate the response time of both algorithms. Figures 6.7 and 6.8 shows the time to initiate the propagation and receive all responses, using both algorithms on several clusters, up to 32 Master Agents running at the same time. On a homogeneous network, our architecture shows good results, with regards to the JXTA overhead, aggregating references of servers collected among 32 Master Agents in less than one second. Note that, under these conditions, most of the trees obtained with the PIF_{async} scheme are stars, the initial propagation from the root reaching other nodes first. In addition, using the PIF_{async} algorithm involves quite few time overhead, in regard of the higher number of messages it generates.

Requests Flooding. We then experimented both algorithms by varying the requests frequency with 15 Master Agents. Figure 6.9 shows the impact of processing multiple requests at the same time inside the graph of MAs, with the same root for every requests. As expected, much better results are obtained by propagating requests with the PIF_{async} algorithm. Using the STAR_{async} algorithm, physical routes used by the JXTA pipes are mostly the same for every requests, strongly increasing

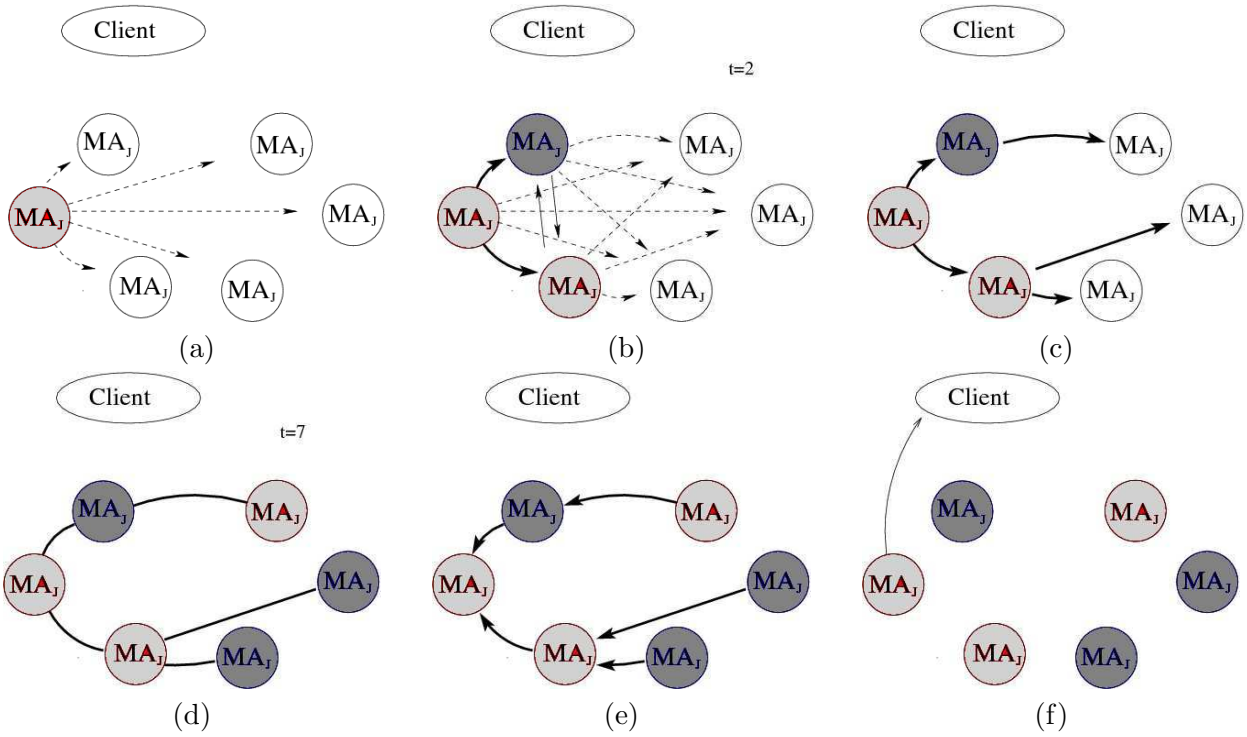


Figure 6.5: PIF-inspired propagation scenario in a DIET multi-hierarchy. The MA that failed to retrieve the requested service in its hierarchy initiates the wave. (a) Some MAs have received the propagated request. They forward it on their turn, and initiate the asynchronous feedback phase (b). All MAs have received the request. A spanning tree has been built (c-d). The feedback phase goes on and ends, responses being on their way to the root and following the links of tree built in the broadcast phase, in a complete asynchronous manner (e). Responses are finally aggregated and sent back to the client (f)

the load on these links. We believe the STAR$_{async}$ algorithm performs so poorly because of the high cost of resolving JXTA pipes, especially when always the same links are stressed. Using the PIF$_{async}$ scheme, logical path and physical routes underneath used during the feedback phase depends on the load of the links during the broadcast phase. Each propagation builds during the broadcast phase a spanning tree used during the feedback phase. The traffic is more distributed and bottlenecks are avoided. The response time remains stable when the frequency of sending becomes high.

Experiments on Overloaded Links. Finally, we simulated a loaded network with loops of `scp` commands (running 13 MAs). Figure 6.10 shows results, varying the number of saturated links around the initiating MA. The STAR$_{async}$ algorithm always uses the saturated links, increasing again the load on the links. The PIF$_{async}$ algorithm allows to avoid most of the traffic around the root by building optimal trees for each request. The response time given by the PIF$_{async}$ scheme is more stable than the STAR$_{async}$ one when the number of overloaded links increases, offering response time similar to those obtained under homogeneous conditions.

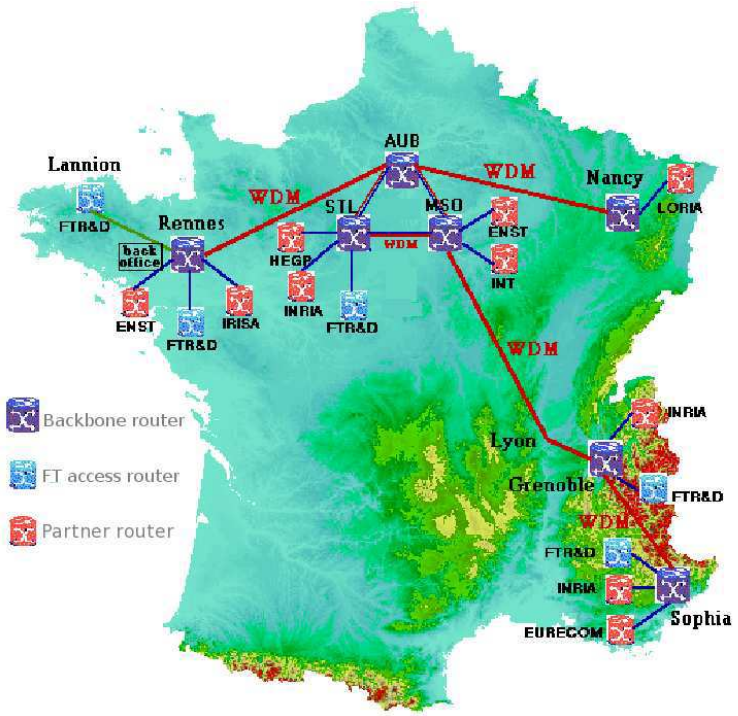


Figure 6.6: The VTHD network.

In this section, we have presented our first application of peer-to-peer concepts in a grid computing service discovery context. By connecting several local hierarchies of service discovery agents, we have greatly improve the scalability of the middleware (removing bottlenecks, increasing the number of services retrieved transparently for clients). Several P2P-fashioned algorithms have been implemented for traversing the architecture presented. Experiments have been conducted showing the efficiency of the middleware built.

6.2 DLPT Prototype Implementation

We here describe the software prototype we developed based on our design presented in Chapter 3. It is written in Java and relies on the JXTA toolbox for low level communications.

The Physical Network. Each physical node of the physical network runs the minimal JXTA peer features to be part of the system. The *physical network* is a set of JXTA peers grouped in a common JXTA group, known under the name *DLPT*. This name is required for each peer to join the system, to be accepted in the group. This name, along with JXTA discovery protocols able to retrieve reachable running peers of this group from a particular JXTA peer given this name, serves as the *out-of-band* mechanism. Once started, any JXTA peer of the DLPT group has a unique ID within the group and can thus receive messages sent by other peers from this group and participate in the management of the tree. A JXTA group is systematically provided with a set of facilities offered by JXTA that the developer needs to overload for his particular scope. For instance primitives for

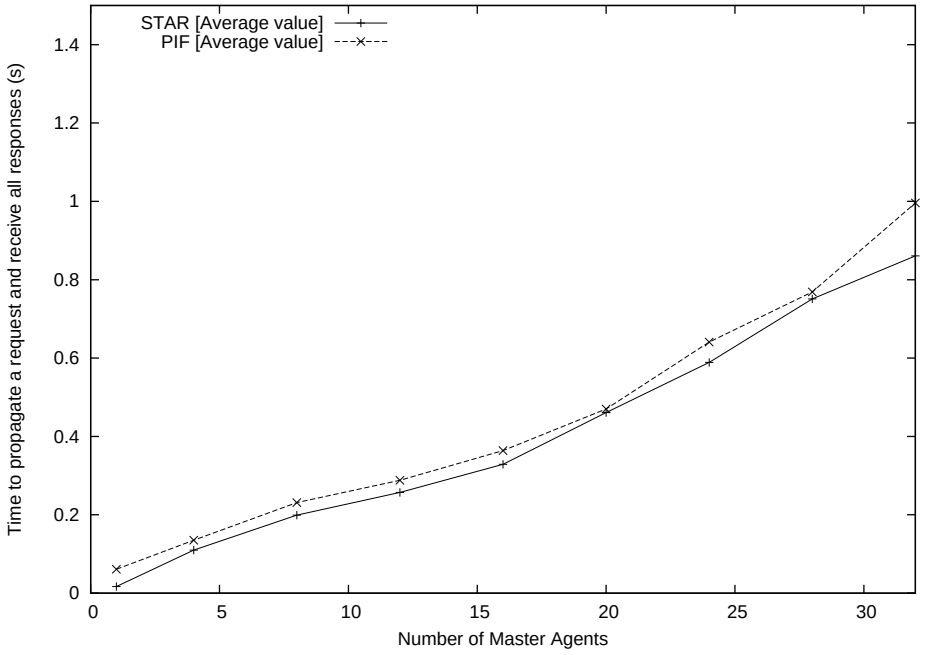


Figure 6.7: Evaluating the cost of using the PIF_{async} compared to the STAR_{async} on one cluster.

sending and receive messages are already written, the developer is just required to write the actions triggered on reception.

The Logical Tree. Each node is implemented as a Java object having fields required from Algorithm 1 from Chapter 3 (on Page 61). Each JXTA peer maintains a list of its own logical nodes. Each logical node also contains the ID of the JXTA peers of its parent and children in the tree, to be able to maintain the tree and make the routing process possible.

The Mapping. The implementation of GetNewProcess() function (refer to Chapter 3, Page 46) calls upon JXTA discovery protocols able to return a random JXTA peer of a given group. Then, the *random* mapping relies on the randomness provided by the JXTA discovery process.

Early Experiments. The tree is built dynamically as some services are declared. This is simulated by JXTA peers when they insert, each of them declaring a set of dummy services. To register a service, a JXTA peer, after having joined the group, contacts a random peer (whose reference is again obtained using the underlying JXTA discovery protocol) and sends a message to it through the underlying JXTA communication protocol. Upon receipt, the JXTA peer decides which of its nodes will initiate the routing in the tree. The routing then follows Algorithm 1, Page 61. The discovery requests are also simulated by a JXTA peer playing the client's role. A first experiment has been conducted on the cluster `capricorne` in Lyon (56 AMD Opteron connected by Gigabit Ethernet through Myrinet-2000 cards). We deployed from 2 up to 25 JXTA peers. Each peer is run on a distinct node of the cluster. Up to 125 services has then been registered. The tree, distributedly constructed by the JXTA peers receiving service registration requests, had a size up to approximately

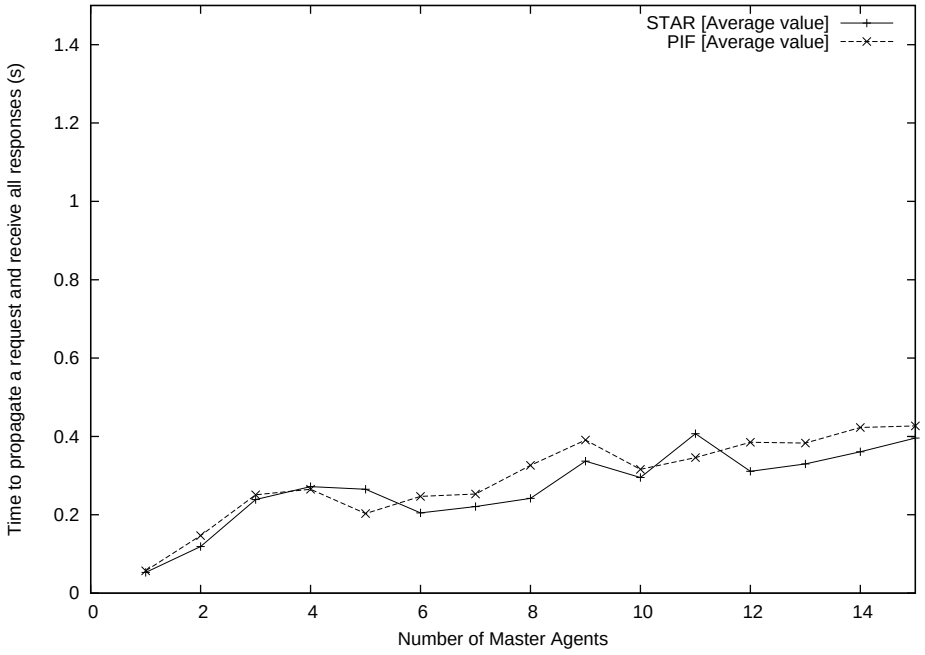


Figure 6.8: Evaluating the cost of using the PIF_{async} compared to the STAR_{async} on two clusters (located in different cities) connected through the VTHD network.

200 nodes. Each peer was running approximately 8 nodes, 5 of which storing information of real services (nodes labeled by *real* keys). We then launched a series of requests to the tree and computed the average time to receive each response once the request transmitted to one peer of the tree. The results are given by Figure 6.11. We see that, when the tree contains 125 services, and thus has a size which is approximately 200 and is distributed among 25 peers, the average response time for a request is approximately 100 milliseconds (ms). Another experiment conducted on the `Grelon` cluster in Nancy (47 AMD Opteron 246 connected by Gigabit Ethernet through Broadcom BCM5721 cards) shows similar performance (see Table 6.1). Again, with a system composed of 47 peers which maintain a tree storing 151 services (and thus has an approximative size of 220), the average response time when requesting the tree is 61 ms. Such preliminary results are very encouraging, as it already shows the capacity of the system to answer to requests very quickly, in spite of an already significant size.

Number of Peers	Number of services	Response time
2	10	21
47	151	61

Table 6.1: DLPT experiment on the `Grelon` cluster, in Nancy.

As we will see in the following section, other experiments have been conducted on 70 nodes reserved beforehand, 14 nodes on 5 different clusters of the Grid'5000 platform.

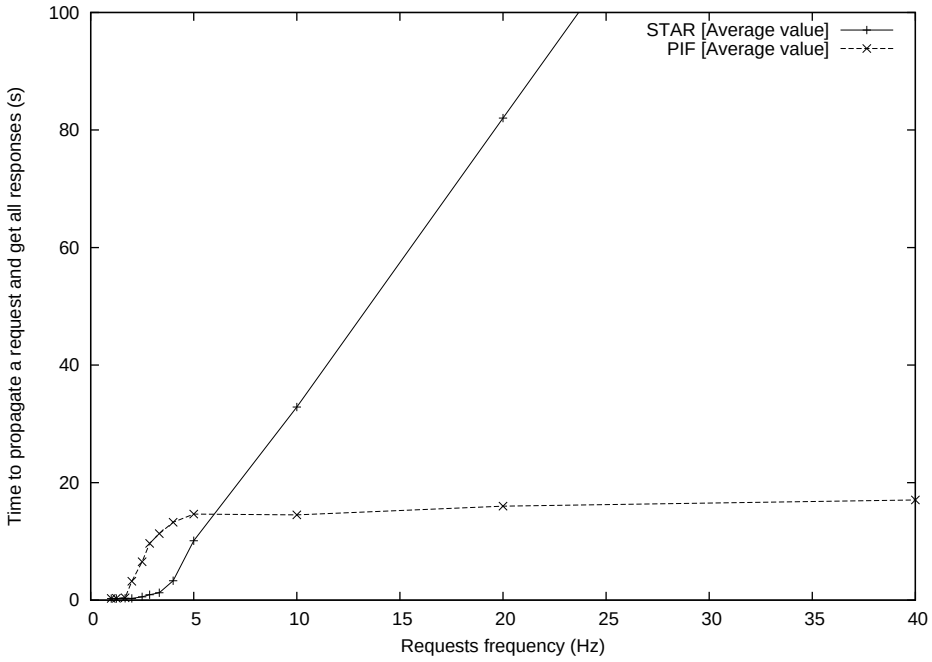


Figure 6.9: Sending 10 requests at various frequencies.

6.3 Using DLPT for Dynamic Virtual Grid Composition

We now present the use of the DLPT and its prototype in the particular context of network-awareness in resource discovery. In Section 6.3, we introduce the reasons of the need of a network-awareness fully decentralized resource discovery system in the context of grid computing, and thus for a decentralized mean to keep information on the performance of the network. In Section 6.3.1, we present the mechanisms added to the DLPT in order to support a network-aware service discovery, the particular way to combine and use DLPT, tackling this new problem. We finally give some preliminary results of our prototype in this particular case in Section 6.3.2.

Context and Motivations

As we highlighted in Chapter 1, in which our aim was the scalability of a grid middleware, the initial context and application environment of our peer-to-peer solution is the grid. Here, we concentrate on enabling the network-awareness of the service discovery process. In other words, we study how to take into account the (dynamic) network performance in the service discovery process, or, more generally, how to be able to *monitor* the network characteristics in a dynamic and decentralized way.

Conceptually, a grid environment (or distributed execution infrastructure) provides the same three fundamental functionalities of a computer: computation, storage, and communication. While the communication services is a kind of "add-on" to the typical stand-alone computer, it is important to understand that the situation is far different with a distributed system like a grid: the communication channels constitute the backbone of the grid and they have a prominent impact on the global and individual performance. Geographically distributed computation and storage units can collaborate efficiently only if they are able to communicate efficiently between each others. Consequently

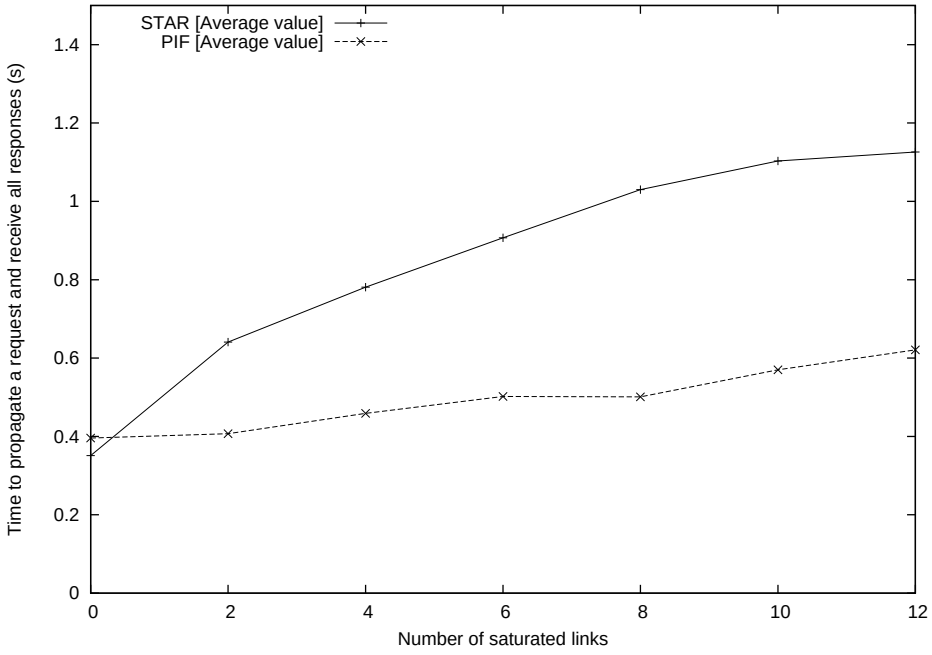


Figure 6.10: Experimenting the PIF_{async} and the STAR_{async} scheme on a network with overloaded links.

monitored or even controlled (end-to-end) communication channels appears to be the required foundation onto which an efficient and viable on-demand dynamic distributed execution infrastructures can be built and used. Services enabling the on-demand or in-advance such aggregation of virtual computing environment meeting the users expectations are envisioned, as for example, in the HIPerNET [94] solution. In classical grid resource discovery systems, the problem of the associated inter-node properties discovery has not been much studied.

With the growing size of distributed network, a flexible and scalable resource discovery approach is still a challenge in service oriented networks. Until here in this dissertation and, as in most papers related to decentralized resource discovery in computational grids– refer to Chapter 2, we focused on computing entities. However, it is increasingly important to consider network attributes such as topology, latency and bandwidth as attributes for resource discovery.

Distributed Applications and Communication Constraints

High-level, structured descriptions of some well-understood Grid network services use cases such as multi-site interactive visualization session or WAN spread High Performance Computing (HPC) are proposed in [60]. They facilitate the identification of network services critical to the Grid middleware and user applications. These scenarios vary in the amount of data that needs to be transfered between nodes, the necessary computing power on each nodes, the latency limitations on inter-nodes communications and the amount of data that needs to be stored (input data as well as output data). They show the various requirements that users can have on resources they reserve.

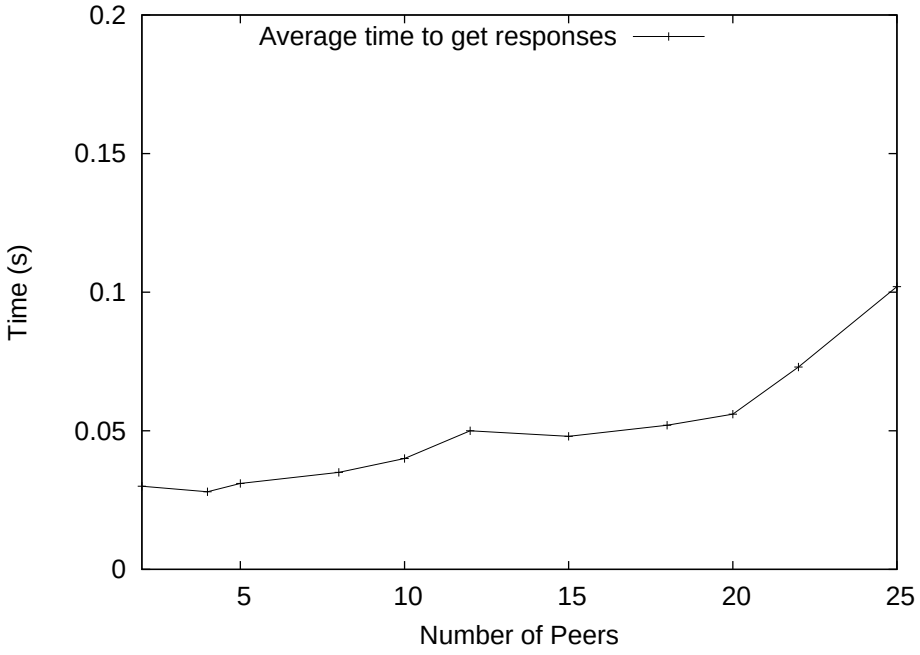


Figure 6.11: Results of DLPT prototype experiment on the `Capricorne` cluster, in Lyon.

Interconnection Network Properties and Services

Given the communication constraints of the demanding applications, one has to consider the available network services that can be combined with the application level services to better select end resources. The abstraction level at which the network properties and services are exposed to the application corresponds to the IP layer (network layer). To consider the properties of the communication channels between the computing resources, the available network service providing them has to be taken into account. Indeed, an end-to-end network service can be with or without quality of service guarantees. Today, IP networks, offering only a best effort packet forwarding service, are exposed to the application through the API socket or through communication libraries via transport protocols that insure reliability and control the sending rates. The applications cannot deal with *advanced reservations* of bandwidth. So if such feature is not proposed explicitly by the network provider, the resource discovery service has to manage implicitly the network, *e.g.,* by selecting end resources that are close or well connected. If there is no possibility to define a virtual topology and a dedicated routing, the user will have to use the physical underlying topology and the default routing. The selection will be made on an estimation or a prediction of end-to-end latency or available bandwidth and has to be transparent to the transport protocol. The transport protocol and its configuration parameters (socket buffer size, number of parallel streams, protocol variant....) may indeed influence the end to end performance, but they are selected and configured by the application (even if a default setting is chosen) so as to adapt to the provided network services (*i.e.,* communication resource).

We consider that a generic network-aware resource discovery service has to be able to deal with explicit bandwidth management service, virtual topology definition service and explicit routing service in order to exploit them when they will be available in future networks. All these services

are network-layer services. By default the current best effort service with default routing and non controlled traffic will be considered. Path characteristics estimation capabilities will be integrated in the discovery service. Consequently there will be three level of network information to expose and make available:

1. characteristics of end-resource communication hardware (number of network interfaces with their type and capacity),
2. characteristics of end-to-end path (end-to-end latency, theoretical capacity of the bottleneck between the two end nodes, end-to-end available bandwidth (can be estimated or predicted),
3. end-to-end QoS guarantee services, *e.g.,* MPLS VPN, diffserv.

Depending on the type of application, there are different kinds of end resources that might have to be tracked in such "capacities" oriented network. Some applications might be compute-intensive; others might require more storage capacities or available bandwidth to transfer the data. There might also be applications whose resource requirements vary during execution (for example a workflow comprising a data transfer phase requiring high bandwidth followed by an intensive computing phase requiring high computing power). Though, such a network might have a lot of resources. Discovering all of them so as to serve the user applications requires special mechanisms.

That's why we need a database containing information about the shared resources that have to be distributed throughout the network on a set of peers, which communicate with each other in a purely-decentralized way to collect, disseminate, index, archive all the resource information and make it visible to as many clients as possible. In that sense, our extension of the DLPT for this purpose presents some analogies with the GMPLS (based on OSPF, RSVP) network layer.

As we will see, this network information are composed of static and of dynamic elements. Some elements can be collected off-line while others are better discovered in-line.

6.3.1 Network-Awareness Using DLPT

The goal of the "network-awareness" is to integrate path characteristics feature discovery in a structured P2P grid resource discovery system. The key idea of this extension can be seen as adaptive combination active probes with multi-attributes queries. Network path characteristics are identified by source and destination locations, end-to-end path delay metrics and end-to-end path throughput metrics as detailed in [98]. In addition, the network information required by application or middleware for resource selection must be simple and coarse grained. Specifying the network topology in detail is very complex and still an open issue [74]. Here, we just want to provide a generic access to network path characteristics comprising classical metrics as well as activable network services. When a new path is established (or modified) by dynamic provisioning, the topology is modified. There are two problems to solve: maintaining the structure dynamically and storing the topology and enabling the access to dynamic information. We now discuss the algorithms themselves.

Building the DLPT Tree

The first type of prefix tree that we will need to maintain is made of strings that are the addresses of grid resources, as shown in Figure 6.12 (DLPT for clustering). The pseudo-code in Algorithm 13 is a simpler version of the general algorithm of construction of the DLPT presented in Chapter 3. We assume several basic functions. ExtractCluster(*addr*) returns the cluster address of a node, by removing the last part

of the address. For instance ExtractCluster(`fr.grid5000.lyon.capricorne.node1`) returns `fr.grid5000.lyon.capricorne`. Prefixes(*addr*) returns the set of sub-domains of *addr*. For instance Prefixes(`fr.grid5000.lyon.capricorne.node27`) returns {`fr`, `fr.grid5000`, `fr.grid5000.lyon`, `fr.grid5000.lyon.capricorne`}. CommonSubDomain($addr_1, addr_2$) returns the greatest common prefix shared by two addresses. For instance, CommonSubDomain(`fr.grid5000.lyon.capricorne.node27`, `fr.grid5000.lyon.sagittaire.node31`) returns `fr.grid5000.lyon`. The UpdateChild message informs a node that it has a new child. On receipt of this message, a node just adds the child provided in the message in its set of children.

We assume that each node is declared once and the addresses are in reverse notation. In the set of possible addresses, given an address a_1, no address a_2 prefixes a_1. When a grid resource registers itself, it sends its address in reverse notation to one tree node (some entry points of the structure may be kept on a webpage, or stored on a server, etc). The address *addr* is packed in a DeviceRegistration message. Upon receipt, a node p computes the cluster part of the address and then processes it according to three cases. First, if the cluster address is the same as the string labeling p, the new device address must be a child of p — refer to Lines 3.03-3.05. Second, if the cluster address is prefixed by the label of p, the new address must be inserted downward in the tree. If we can find a child of p also prefixing *addr*, the request moves to this child. Otherwise, the cluster and the device are inserted as a subtree of p — refer to Lines 3.06-3.12. Finally, if none of the previous case are satisfied, we must study the greatest common subdomain of p and *addr*. If the parent of p shares the same common subdomain with *addr* as p, the request moves to f_p. Otherwise, we must create an intermediary node reflecting this relation *i.e.,* the father of both the cluster of *addr* and p — refer to Lines 3.13-3.23.

Retrieving addresses of all nodes of a particular cluster relies on algorithms of Chapter 3.

Storing Numerical Characteristics

Dealing with numerical range queries is useful when dealing with for instance CPU, bandwidth or latency. A *numerical* tree may look like the example on Figure 6.12 (*e.g.,* DLPT for CPU power). The numbers declared have a constant size, possibly padded with zeros. For instance, if we assume a length of 3, the number 53 will be processed as 053. Then, to retrieve objects corresponding to a value between 130 and 350, the range query can start on the root node of the example tree. Then the root decides to which of their children it will send the query depending on whether they are susceptible to have some nodes in their subtree pertained by the range. Repeating this test at every level, the request is sent to nodes 1, 15, 153, 155, 158, 2, 207, 245, 3, and 321. Then, the leaf nodes send the information they store *e.g.,* addresses of nodes satisfying this range back to the initiator of the request.

Requests Definition

We saw previously a few Grid services use cases that allow us to describe the type of requests our system processes:

- Computing clusters are defined by the computing power they can offer and the interconnection between their nodes. The 3 attributes needed to modelize the computing power are the *amount of nodes* in the cluster the *computing power* of a node itself (in flops) and the *maximal CPU load* on all of the these nodes. The required *bandwidth* and *latency* between the nodes are

Algorithm 13 Routing and inserting a device registration, on node p

```
2.01  Variables:  p, identifier of p
                  f_p, identifier of the father of p
                  C_p, finite set of children identifiers of p

3.01  upon receipt of <DeviceRegistration, addr> from
3.02 do     clus =ExtractCluster(addr)
3.03        if clus = p then
3.04              NewNode(addr, p, ∅))
3.05              C_p = C_p ∪ {addr}
3.06        elseif p ∈ Prefixes (clus) then
3.07              if ∃q ∈ C_p : | CommonSubDomain(clus, q)| > |CommonSubDomain(clus, p)| then
3.08                    send(<DeviceRegistration, addr>, q)
3.09              else
3.10                    NewNode(clus, p, {addr})
3.11                    NewNode(addr, clus, ∅))
3.12              C_p = C_p ∪ {clus}
3.13        else
3.14              common = CommonSubDomain(addr, p)
3.15              if (f_p ≠ ⊥) ∧ (common =CommonSubDomain(addr, f_p)) then
3.16                    send(<DeviceRegistration, addr>, f_p)
3.17              else
3.18                          NewNode(common, f_p, {clus, p})
3.19                    if (f_p ≠ ⊥) then
3.20                          send(<UpdateChild, (p, common)>, f_p)
3.21                    NewNode(clus, common, {addr})
3.22                    NewNode(addr, clus, ∅)
3.23                    f_p := common
```

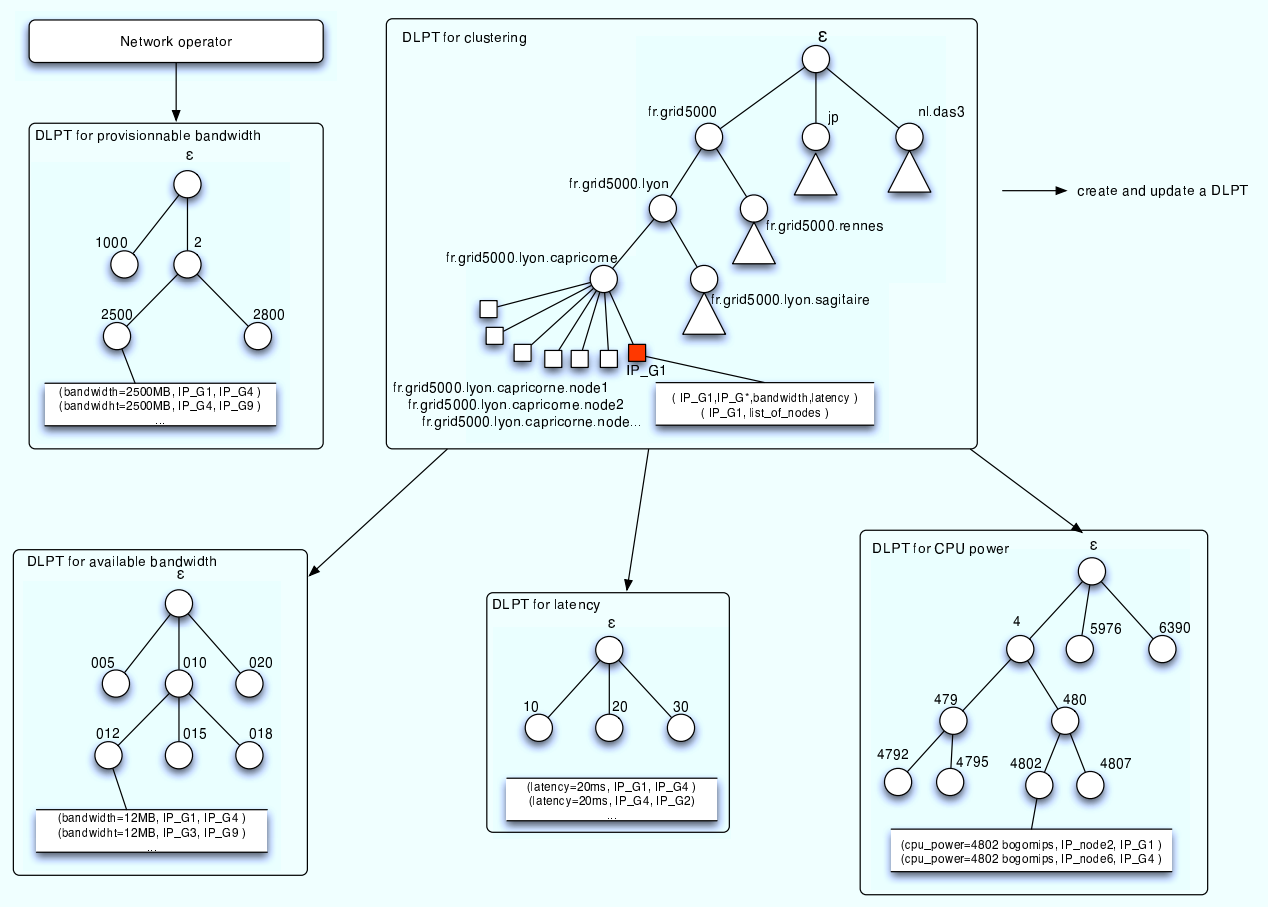


Figure 6.12: Example of an architecture using multi-DLPT approach for network-aware service discovery.

used to define the interconnection. A typical query would be a cluster of 15 nodes with a CPU load inferior to 10 percent, where all the nodes are interconnected with links of at least 1Gb bandwidth and at most 0.1ms latency.

- Storage spaces are defined by the storage space they offer. A query can be as simple as a *total storage space* expressed in gigabytes, terabytes, in this case the user is not interested in the amount of nodes on which the data may be spread. It can also be more elaborate and specify a *maximum amount of nodes* on which the data is shared, as well as the interconnection (*bandwidth* and *latency*) between these nodes.
- Interconnection links are defined by a *bandwidth* and a *latency*. They are used to describe links between different clusters or storage spaces. For example if a user wants to reserve a storage space on one hand and a computing cluster on the other hand, he will as well describe the link interconnecting these 2 entities.

Furthermore, our system is locality-aware thanks to its internal DNS-like structure and thus a user can reserve resources located at a certain spot. This can be useful if he wants to co-locate certain resources such as in the visualization session case where the rendering cluster needs to be located at the display site.

Note that if a user requests a cluster of n nodes it does not matter whether the nodes are actually on the same physical cluster as long as the nodes forming it and their interconnection match the user's requirements.

Resources Description

Extremity Resource Description. The extremity resource description has to be designed so that the system is able to answer queries formulated as described above. The minimal amount of information needed in this purpose for each node is (i) Name: a unique name inside the cluster. Note that it will also be unique in the whole system given the cluster, site and all other hierarchically higher entity names are also unique. (ii) Locator: IP address of the node, used to communicate with it. (iii) Computing power, measured in flops. (iv) Load: current load of the CPU. This is a dynamic parameter that varies over time between 0 and 100 percents. It has to be monitored. (v) Storage space: amount of space locally available for storage. This is also a dynamic parameter that has to be monitored. (vi) capacity of the network interface. This can give a higher bound on the rate at which this node can exchange data. In intra-cluster communication this can be considered as the actual rate.

Network Resource Description. The two main parameters to describe the network resources are latency and bandwidth of the links. Storing the whole topology is far from being possible. Indeed a N-node network would yield $O(N^2)$ pairs of node and twice as many measures, which is totally not scalable. To lower the amount of data that needs to be stored we will consider that a link connecting a node n_A in a physical cluster A with a node n_B in a physical cluster B will have the same characteristics for all n_A and n_B. We thus randomly elect one *cluster representative node* per cluster that takes care of measuring and monitoring the link characteristics between its own cluster and others. The latency can be measured with a simple `ping` while the available bandwidth requires more elaborate tools such as `pathload` [87, 115]. The cluster representative will store a table with one entry for each other cluster filled with actual values for bandwidth and latency of the link interconnecting the two for them.

Processing the Requests

We end the description of the system by giving some examples of how the previously specified queries can be processed by DLPT trees. We assume we have the address-tree, whose construction was detailed previously and illustrated in Figure 6.12. We refer to this tree as the **topology DLPT**. The topology tree is built, in the classical way, as resources are registered.

We also assume that we have numerical DLPT trees as the one illustrated in **available bandwidth DLPT** (for instance), see Figure 6.12. We use the multi-DLPT to store the following information:

- **CPU**. Expressed for instance by BogoMips, a client may specify a minimum value of the resource it seeks. The information stored on the node 10 is the set of addresses whose corresponding node's CPU is 10 BogoMips. This DLPT tree is also built as resources register, giving their CPUs.

- **Latency**. A user may want to use a network link whose latency is less than a given value. We assume the latency is computed between two clusters. Then the information stored on the node 5 is the set of clusters pairs whose link latency is 5 ms.

- **Available Bandwidth**. In a best-effort policy, a user may want to reserve a network link whose bandwidth is higher than a given value. The information stored is similar to the one stored on the nodes of the latency DLPT tree.

- **Provisionnable Bandwidth**. An operator may want to sell some network resources, in which case, clients may want to dynamically know what they can buy. So we use another DLPT tree storing dynamically the provisionnable bandwidth.

The last three trees are built when clusters exchange messages. It happens when clusters discover each other by using the topology tree. This allows them to store the network information and to remain efficient in terms of delay of request processing. Indeed, all the dynamic information of the network could be stored in the topology tree, but it would lead to flood the entire tree to retrieve links satisfying a requested property. So as to accelerate the processing of such queries we keep these last three numerical trees updated. Hence the good properties of lexicographic trees are maintained.

We now give an example of processing a complex query. A client wants to discover 20 nodes on the network where CPU power is greater than 100 BogoMips, the available bandwidth of the link between two clusters of these nodes is superior to 1 Gbps and the latency of this link is less than 1 ms.

The client interface sends a first request to a random node in each of the *available bandwidth* and *latency* trees, which returns to the client a set of clusters pairs satisfying the requested values.

In parallel, a second request is sent to the CPU tree to get registered nodes whose CPU is greater than 100.

Finally, a request to the topology tree returns several sets of 20 nodes chosen among pair of clusters returned by the first request. The final results can be obtained by performing a simple join operation to keep only the sets of nodes that contain nodes whose CPU power is greater than 100. Note that this request would actually yield a very large amount of results in a real case, they may though be limited to the n first values by simply decrementing a counter as the request travels through the DLPTs and responses are appended.

6.3.2 Results

We have developed a prototype of the solution we proposed for a network-aware resource discovery infrastructure to evaluate its feasibility and the technical issues it can raise. This prototype has been tested on Grid'5000 [33], a nation-wide infrastructure gathering about 5000 CPUs dedicated to research purposes. Grid'5000 provides a deep reconfiguration mechanism allowing researchers to deploy, install, boot and run their specific software images, possibly including all the layers of the software stack. In a typical usage sequence, a user reserves a partition of Grid'5000, deploys its software image on it, runs the experiment, collects results and relieves the machines. This reconfiguration capability allows all users to run their applications in the software environment exactly corresponding to their needs.

The experimentation is based on the implementation of the concepts presented in this section and integrated to the prototype described in Section 6.2. Experimentations rely on a single DLPT tree where a node is defined by its name (used for indexing it in the tree), its IP address, its network interfaces and its cluster id being part of the information contained in the *value* of the service when registerd. To evaluate the proposed solution, we have elaborated a scenario that allows a user to reserve a cluster. It can define the same parameters in his requests as the ones used to describe a resource stored in the implemented DLPT. The experiment has been conducted on a total of 70 nodes reserved beforehand, 14 nodes on 5 different clusters (`capricorne` in Lyon, `grillon` in Nancy, `bordeplage` in Bordeaux, `parasol` in Rennes and `gdx` in Orsay).

Recall that resources simply have to send a registration request containing a description of itself to a DLPT peer to register. Once enough resources are registered, a client can reserve them. We can for example request nodes in a same cluster to ensure a lower latency between them. Tests shown that by specifying a cluster name we can achieve an average latency of 0.096ms whereas it was equal to 7.021ms for nodes dispatched in several different clusters. This can thus be a useful feature for users in need of a *tight bag* of nodes *i.e.,* a set of nodes interconnected with low latency links. Below are displayed results of a ping test from one machine (172.28.54.29) to other machines reserved in the same cluster.

```
PING 172.28.54.36 (172.28.54.36) 56(84) bytes of data.
rtt min/avg/max/mdev = 0.088/0.096/0.102/0.014 ms
PING 172.28.54.38 (172.28.54.38) 56(84) bytes of data.
rtt min/avg/max/mdev = 0.089/0.093/0.103/0.011 ms
PING 172.28.54.42 (172.28.54.42) 56(84) bytes of data.
rtt min/avg/max/mdev = 0.090/0.097/0.109/0.011 ms
PING 172.28.54.44 (172.28.54.44) 56(84) bytes of data.
rtt min/avg/max/mdev = 0.091/0.097/0.106/0.009 ms
PING 172.28.54.45 (172.28.54.45) 56(84) bytes of data.
rtt min/avg/max/mdev = 0.089/0.095/0.102/0.009 ms
```

Below are results of a ping test from one machine (172.28.54.29) to other machines reserved in different clusters.

```
PING 131.254.202.112 (131.254.202.112) 56(84) bytes of data.
rtt min/avg/max/mdev = 11.641/11.653/11.671/0.012 ms
PING 131.254.202.115 (131.254.202.115) 56(84) bytes of data.
rtt min/avg/max/mdev = 11.627/11.630/11.634/0.136 ms
PING 131.254.202.116 (131.254.202.116) 56(84) bytes of data.
rtt min/avg/max/mdev = 11.624/11.628/11.642/0.118 ms
PING 172.28.54.42 (172.28.54.42) 56(84) bytes of data.
rtt min/avg/max/mdev = 0.089/0.098/0.117/0.012 ms
PING 172.28.54.44 (172.28.54.44) 56(84) bytes of data.
rtt min/avg/max/mdev = 0.090/0.098/0.103/0.013 ms
```

Tests using `iperf` have also shown that intra-cluster bandwidth is much higher than inter-cluster bandwidth. Tests were conducted with default buffer and window size configured in the Linux kernel and without any TCP tuning. If all the nodes are requested in a same cluster the average bandwidth between all of them can be as high as 940Mbit/s (for nodes with a 1Gbps interface), while inter-

cluster bandwidth will be limited to less than a hundred Mbit/s between certain clusters even with nodes carrying a 1Gbps interface. We conducted the test with the same reservations as previous test. First part of the test from the same machine (172.28.54.29) to machines in a same cluster using iperf with 1000 MBytes file to transmit. Results show a very high intra-cluster bandwidth:

```
[  4] local 172.28.54.29 port 5001 connected with 172.28.54.42 port 45485
[  4]  0.0- 8.9 sec  1000 MBytes    941 Mbits/sec
[  4] local 172.28.54.29 port 5001 connected with 172.28.54.41 port 33802
[  4]  0.0- 8.9 sec  1000 MBytes    941 Mbits/sec
[  4] local 172.28.54.29 port 5001 connected with 172.28.54.44 port 59204
[  4]  0.0- 8.9 sec  1000 MBytes    941 Mbits/sec
[  4] local 172.28.54.29 port 5001 connected with 172.28.54.40 port 46698
[  4]  0.0- 8.9 sec  1000 MBytes    941 Mbits/sec
```

Second part of the test was conducted from the same machine (172.28.54.29) to machines reserved in different clusters with same conditions (iperf with a 1000 Mbytes file). Results show very low bandwidth in inter-cluster connection:

```
[  4] local 172.28.54.29 port 5001 connected with 172.28.54.44 port 54726
[  4]  0.0- 8.9 sec  1000 MBytes    941 Mbits/sec
[  4] local 172.28.54.29 port 5001 connected with 172.28.54.42 port 34750
[  4]  0.0- 8.9 sec  1000 MBytes    941 Mbits/sec
[  4] local 172.28.54.29 port 5001 connected with 131.254.202.112 port 59197
[  5] local 172.28.54.29 port 5001 connected with 131.254.202.115 port 44866
[  6] local 172.28.54.29 port 5001 connected with 131.254.202.116 port 36177
[  4]  0.0-124.9 sec  1000 MBytes  67.2 Mbits/sec
[  5]  0.0-124.9 sec  1000 MBytes  67.2 Mbits/sec
[  6]  0.0-125.1 sec  1000 MBytes  67.0 Mbits/sec
```

Further results show that specifying the capacity of the nodes' network interface is not enough to guarantee an intra-cluster bandwidth except if the nodes are reserved on a same physical cluster. Although most of the sites in Grid'5000 are interconnected with 10Gb links, the capacity of these links is shared between many users at a time. By only specifying the capacity of the nodes' network interface we actually rely on the best effort service. Grid'5000 does not offer any link provisioning capability. If one was to be implemented, further works may interact with it to provision links on demand and thus create virtual clusters spanning multiple sites totally transparently to the user with bandwidth and latency comparable with the ones measured between nodes of a same physical cluster.

6.4 Conclusion

In this chapter, we have first presented the peer-to-peer extension of DIET, the algorithms implemented inside, and experimentation results, showing a first efficient step to the integration of peer-to-peer technologies into grid middleware. Then, we have presented the prototype implementation of the DLPT architecture. This software is again based on the JXTA toolbox for low level communications. Early experimental results, conducted on clusters of the Grid'5000 platform shows very small latencies to receive responses to a discovery request as the tree and underlying network grow. This results are very promising and open the door for a deeper experimentation. As a use-case and application, and with the goal to provide an infrastructure offering network-aware resource discovery in a very widely distributed way, we extended the DLPT and our prototype. It was shaped according to the user needs drafted from the most common use cases encountered in grid environments. It was furthermore designed in regards of the latest technologies that are to emerge in the following years such as bandwidth provisioning and network virtualization features. The prototype developed to attest the most basic functionalities of this infrastructure gave encouraging results.

Conclusion and Future Works

Conclusion

Grid computing aims at gathering geographically heterogeneous computing resources. Within such platforms, servers provide computing services to some clients. A client looking for a particular type of service needs to discover the services available on the platform to find a service matching its requirements. The process matching clients' needs and server's offers is the *Service Discovery*. Among the barriers still hindering the deployment of computational grids over large scale platforms, we find several problems related to service discovery. These problems are mainly related to its scale, its efficiency over heterogeneous platforms, and its ability to face the dynamic nature of the platform. Peer-to-peer technologies traditionally address these aspects. The peer-to-peer systems became a new field of investigation for designers of grid middleware.

In this dissertation, we have presented different aspects of a peer-to-peer fashioned solution for the service discovery issue in computational grids. The purpose of the first Chapter was to ease the understanding of the basic components of the problem and to propose a clear statement of it. In the same chapter, we have presented the environment of our work, namely computational grids, the traditional approaches to service discovery within them, and put our problem in more precise words.

As our solution belongs to a series of works relying on peer-to-peer principles, a first important part of our background and related works come from the peer-to-peer area. Also, as our solution for fault-tolerance uses self-stabilization concepts, the second main part of our background comes from the self-stabilization area. In Chapter 2, we have given the required background allowing to understand our solution and compare it with similar works, on relevant points (complexities, scale, efficiency according to several parameters, load balancing, fault-tolerance.)

In Chapter 3, we have presented the basic design of a solution (called $DLPT$) to the service discovery problem. We have justified the choice of the structure used and we have given a set of algorithms ready to be implemented, based on the message passing paradigm, for the construction and maintenance of such an architecture. A study of its complexities and advantages relative to very similar approaches have been discussed, and simulation results presented to assert its relevance and performance.

In Chapter 4, we have focused on mapping and load balancing issues, which were remaining drawbacks of the initial solution, as discussed at the end of Chapter 3. The contribution in Chapter 4 are twofold. First, we reduce the global cost of our architecture (in terms of maintenance and amount of effective communications required) by a self-contained mapping scheme, naturally clustering entities of our logical system on processors of the network. Second, we provide a load balancing heuristic whose objective function (maximizing the throughput) is slightly different from other approaches in literature, making it more efficient for practical systems. Finally, by simulations, both the architecture and the heuristic are validated. As detailed at the end of the chapter, this last step allows to

better see how our solution outperforms related works.

Chapter 5 deals with alternative solutions to the fault-tolerance issue of this kind of systems. We provide mechanisms based on self-stabilization. Our point was to provide mechanisms able to maintain (or automatically rebuild) a consistent system after traditional proactive approaches, based on costly replication, have failed. The contribution are here threefold. First, we provide a protocol able to reconnect disconnected subparts of the system after an arbitrary number of crashes. In the way to a fully self-stabilizing protocol, we have then proposed a *snap*-stabilizing algorithm, which, as provided by the self-stabilization paradigm, maintains our architecture after crashes, wrong initializations, or any transient fault affecting the system. This second protocol, although optimal in stabilization time, has been written in a coarse-grained model, abstracting details of message exchanges. As a last contribution, we have written a self-stabilizing protocol able to constantly automatically rebuild our architecture starting from any arbitrary configurations. Relying on the message-passing paradigm, this last protocol is ready to be implemented on real platforms. Again, complexities and simulation results allow to give a good idea of the performance of our algorithms.

Finally, in Chapter 6, we discuss our practical contributions. In a first work related to service discovery in a grid computing environment, and conducted in 2004 as a preliminary of this thesis, we have extended an existing middleware, DIET, in a peer-to-peer fashion for its scalability. Real experiments conducted over several connected clusters have shown the efficiency of the approach. This extension is part of the DIET software package, available on the web[4]. Our second practical work deals with the development of a prototype implementation of our DLPT solution and the study of its use in a practical context of the reservation and provisioning of network resources. Early experiments have been conducted on the Grid'5000 platform. Very promising results have been obtained, which already shows the viability of our implementation.

Future Work

On our solution itself, several improvements can be undertaken.

- The replication process and its implications on the performance, dealing with load balancing and topology awareness have only been mentioned in Chapter 3 (Page 50). A deeper study is here required to see the real impact of the replication solution, both in terms of topology awareness and load balancing and how to implement the right trade-off between these two objectives.

- Still in Chapter 3, we describe potential cache mechanisms (Page 52) in order to optimize and balance load for hot-spots. Even if a complete solution for the load-balancing issue is given in Chapter 4, a simulation study of the impact of this cache mechanisms has been conducted only on the reduction of lookup length but not on the load balancing itself. Further simulation investigations could help to see the relevance of such caching schemes.

- The design of our load balancing heuristic presented in Chapter 4 differs from other approaches by its objective function, which focuses on the maximization of the throughput. However, it could be interesting to improve it by coupling it to a random choice approach. The load balancing process is only locally executed, between two neighbors. Randomly choosing several potential partners for load balancing could be a new way to improve the complete process.

[4] http://graal.ens-lyon.fr/DIET

Finally, the next step is also to implement these features inside the prototype and experiment them.

- At the end of Chapter 5, we measure by simulation the *efficiency* of our last self-stabilizing protocol, from the *client's point of view*. We study the gain on the amount of discovery requests that were satisfied by using the self-stabilizing protocol (even when it is not able to completely rebuild the structure in the case of a too high failure rate). This study can be completed, for instance by opening investigations on the level of graceful degradation that can be ensured in the system, when varying the frequency of failures, in particular from the client's point of view. In the same order of ideas, the *optimality* for the client would be that all requests are satisfied. The schemes that we presented does not ensure it, since maintenance and discovery are two distinct processes. Another interesting point could be to reach this optimality, and study its cost. (It will inevitably requires mixing the two processes, and delaying the responses to the clients.)

- Still dealing with self-stabilization, another interesting work would be to make a snap-stabilizing version of our message-passing self-stabilizing protocol. Recent works [50], provide new tools to achieve snap-stabilization in such environments. Finally, an implementation of all the provided fault-tolerance mechanisms in the prototype is planned.

In a more global vision, such service discovery systems are only a part of future grid middleware. The service discovery, as a part of the whole middleware, will need, for instance, to communicate with the scheduling part of the middleware. Here, the scheduling scope is to refine the set of services matching the client's requirements, by choosing the service which satisfies the scheduling policy when an important number of services are requested by several clients at the same time. The scheduling problem in a P2P environment relies on decentralized scheduling and involve for instance task migration [19].

Moreover, in emerging grid platforms, resources are made available through a batch scheduler system, like OAR [39] and reserved through a given policy, most of the time first-come-first-served (FCFS). The batch reservation introduces a new kind of volatility. A node can become unusable not because it crashed, but simply because the reservation has ended. Integrating these three components (service discovery, batch reservation, scheduling) will require to define protocols between these entities for putting it all together. This will be a step to the global *integrated* middleware.

Finally, with the emergence of increasingly powerful infrastructures, like *Petascale computers* [25], the interconnection of large research grids [1, 140, 143] or incredibly large desktop grids [107], researchers are thinking to the next step: interconnecting these highly heterogeneous platforms in a single environment. Providing services inside such an environment to users is a very challenging issue, in which Service Discovery will be a fundamental component.

List of Figures

References

[1] Grid 5000 project. http://www.grid5000.org.

[2] Lhc grid computing project. http://lcg.web.cern.ch/lcg/, year = 2008.

[3] Reference Model for Service Oriented Architecture 1.0. Technical report, OASIS, 2006.

[4] Lhc (large hadron collider). http://public.web.cern.ch/public/en/LHC/LHC-en.html, 2008.

[5] Open Grid Forum. http://www.ogf.org, 2008.

[6] S3L Library, Sun Microsystems. http://dlc.sun.com/pdf/816-0653-10/816-0653-10.pdf, 2008.

[7] ABC@home. http://abcathome.com, 2008.

[8] K. Aberer, A. Datta, and M. Hauswirth. Multifaceted Simultaneous Load Balancing in DHT-Based P2P Systems: A New Game with Old Balls and Bins. In *Self-star Properties in Complex Information Systems*, pages 373–391, 2005.

[9] Y. Afek, S. Kutten, and M. Yung. Memory-Efficient Self Stabilizing Protocols for General Networks. In *WDAG*, pages 15–28, 1990.

[10] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services: Concepts, Architecture and Applications*. Springer Verlag, 2004.

[11] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic Local Alignment Search Tool. *J. Mol. Biol.*, 215(3):403–410, October 1990.

[12] P. Amestoy, M. Daydé, C. Hamerling, M. Pantel, and C. Puglisi. Management of Services Based on a Semantic Description Within the GRID-TLSE Project. In *High Performance Computing for Computational Science - VECPAR 2006*, volume 4395 of *LNCS*, pages 634–643. Springer-Verlag Berlin Heidelberg, 2007.

[13] P. R. Amestoy, I. S. Duff, J. Koster, and J.-Y. L'Excellent. MUMPS: A Multifrontal Massively Parallel Solver. *ERCIM News*, 50:14–15, July 2002. European Research Consortium for Informatics and Mathematics (ERCIM), http://www.ercim.org.

[14] D. Anderson. BOINC: A System for Public-Resource Computing and Storage. In *GRID'04: Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing*, pages 4–10, Washington, DC, USA, 2004. IEEE Computer Society.

[15] D. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. SETI@home: an Experiment in Public-Resource Computing. *Commun. ACM*, 45(11):56–61, 2002.

[16] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. SIAM, Society for Industrial and Applied Mathematics, Philadelphia, 1999.

[17] A. Andrzejak and Z. Xu. Scalable, Efficient Range Queries for Grid Information Services. In *Peer-to-Peer Computing*, pages 33–40, 2002.

[18] A. Arora and M.G. Gouda. Distributed Reset. *IEEE Trans. Computers*, 43(9):1026–1038, 1994.

[19] M. Arora, S. K. Das, and R. Biswas. A De-Centralized Scheduling and Load Balancing Algorithm for Heterogeneous Grid Environments. In *ICPP Workshops*, pages 499–505, 2002.

[20] T. Asano, D. Ranjan, T. Roos, E. Welzl, and P. Widmayer. Space-Filling Curves and their Use in the Design of Geometric Data Structures. *Theoretical Computer Science*, 181, 1997.

[21] J. Aspnes and G. Shah. Skip graphs. *ACM Trans. Algorithms*, 3(4):37, 2007.

[22] B. Awerbuch, S. Kutten, Y. Mansour, B. Patt-Shamir, and G. Varghese. Time Optimal Self-stabilizing Synchronization. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing (STOC '93)*, pages 652–661, 1993.

[23] B. Awerbuch, B. Patt-Shamir, and G. Varghese. Self-Stabilizing End-to-End Communication. *Journal of High Speed Networks*, 5(4):365–381, 1996.

[24] Y. Azar, A. Z. Broder, A. R. Karlin, and E. Upfal. Balanced Allocations. *SIAM J. Comput.*, 29(1):180–200, 2000.

[25] D. Bader. *Petascale Computing Algorithms and Applications*. Chapman and Hall/CRC, 2007. ISBN 1-58488-909-8.

[26] S. Basu, S. Banerjee, P. Sharma, and S. Lee. NodeWiz: Peer-to-Peer Resource Discovery for Grids. In *5th International Workshop on Global and Peer-to-Peer Computing (GP2PC)*, May 2005.

[27] D. B. Batchelor. Fusion - High Performance Computing in Magnetic Fusion Energy Research. In *SC*, 2006.

[28] R. Bayer. Binary B-Trees for Virtual Memory. In *SIGFIDET Workshop*, pages 219–235, 1971.

[29] D. Bein, A. K. Datta, and Villain V. Snap-Stabilizing Optimal Binary Search Tree. In Springer LNCS 3764, editor, *Proceedings of the 7th International Symposium on Self-Stabilizing Systems (SSS'05)*, pages 1–17, 2005.

[30] A. Bharambe, M. Agrawal, and S. Seshan. Mercury: Supporting Scalable Multi-Attribute Range Queries. In *Proceedings of the SIGCOMM Symposium*, August 2004.

[31] M. Bienkowski, M. Korzeniowski, and F. Meyer auf der Heide. Dynamic Load Balancing in Distributed Hash Tables. In *IPTPS*, pages 217–225, 2005.

[32] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. 1997.

[33] R. Bolze, F. Cappello, E. Caron, M. Daydé, F. Desprez, E. Jeannot, Y. Jégou, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, P. Primet, B. Quetier, O. Richard, E. Talbi, and Touché I. Grid'5000: a Large Scale and Highly Reconfigurable Experimental Grid Testbed. *International Journal of High Performance Computing Applications*, 20(4):481–494, November 2006.

[34] A. Bui, A. Datta, F. Petit, and V. Villain. State-Optimal Snap-Stabilizing PIF in Tree Networks. In IEEE, editor, *Proceedings of the 4th International Workshop on Self-Stabilizing Systems*, pages 78–85, 1999.

[35] J. W. Byers, J. Considine, and M. Mitzenmacher. Simple Load Balancing for Distributed Hash Tables. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS'03)*, pages 80–87, 2003.

[36] M. Cai, M. R. Frank, J. Chen, and P. A. Szekely. MAAN: A Multi-Attribute Addressable Network for Grid Information Services. *Journal of Grid Computing*, 2(1):3–14, 2004.

[37] Y. Caniou, E. Caron, G. Charrier, A. Chis, F. Desprez, and E. Maisonnave. Ocean-Atmosphere Modelization over the Grid. In *The 37th International Conference on Parallel Processing (ICPP 2008)*, Portland, USA, 2008. IEEE. To appear.

[38] Y. Caniou, E. Caron, H. Courtois, B. Depardon, and R. Teyssier. Cosmological Simulations using Grid Middleware. In *Fourth High-Performance Grid Computing Workshop (HPGC'07)*, Long Beach, California, USA, March 2007. IEEE.

[39] N. Capit, G. Da Costa, Y. Georgiou, G. Huard, C. Martin, G. Mounié, P. Neyron, and O. Richard. A Batch Scheduler with High Level Components. In *CCGRID*, pages 776–783, 2005.

[40] E. Caron and F. Desprez. DIET: A Scalable Toolbox to Build Network Enabled Servers on the Grid. *International Journal of High Performance Computing Applications*, 20(3):335–352, 2006.

[41] E. Caron, F. Desprez, and G. Le Mahec. Parallelization and Distribution Strategies of Large Bioinformatics Requests over the Grid. In *ICA3PP*, pages 257–260, 2008.

[42] E. Caron, F. Desprez, F. Petit, and V. Villain. A Hierarchical Resource Reservation Algorithm for Network Enabled Servers. In *IPDPS'03. The 17th International Parallel and Distributed Processing Symposium*, Nice, France, April 2003.

[43] P. Chan and D. Abramson. A Scalable and Efficient Prefix-Based Lookup Mechanism for Large-Scale Grids. In *3rd IEEE International Conference on e-Science and Grid Computing, e-Science 2007*, Bangalore, India, December 2007. IEEE.

[44] R. Chand, M. Cosnard, and L. Liquori. Powerful Resource Discovery for Arigatoni Overlay Network. *Future Generation Comp. Syst.*, 24(1):31–38, 2008.

[45] E.J.H. Chang. Echo Algorithms: Depth Parallel Operations on General Graphs. *IEEE Trans. on Software Engineering*, SE-8:391–401, 1982.

[46] N.S. Chen, H.P. Yu, and S.T. Huang. A Self-stabilizing Algorithm for Constructing Spanning Trees. *Information Processing Letters*, 39:147–151, 1991.

[47] A. Cournier, A. K. Datta, F. Petit, and V. Villain. Enabling Snap-Stabilization. In *ICDCS'03: Proceedings of the 23rd International Conference on Distributed Computing Systems*, Washington, DC, USA, 2003. IEEE Computer Society.

[48] A. Datta, M. Hauswirth, R. John, R. Schmidt, and K. Aberer. Range Queries in Trie-Structured Overlays. In *The Fifth IEEE International Conference on Peer-to-Peer Computing*, 2005.

[49] A. K. Datta, M. Gradinariu, M. Raynal, and G. Simon. Anonymous publish/subscribe in P2P networks. In *The 17th International Parallel and Distributed Processing Symposium, IPDPS'03*, 2003.

[50] S. Delaët, S. Devismes, M. Nesterenko, and S. Tixeuil. Brief announcement: Snap-stabilization in message-passing systems. In *Principles of Distributed Computing (PODC 2008)*, August 2008.

[51] J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, X. S. Li, and J. W. H. Liu. A Supernodal Approach to Sparse Partial Pivoting. *SIAM J. Matrix Analysis and Applications*, 20(3):720–755, 1999.

[52] The Bamboo DHT. http://bamboo-dht.org/, 2008.

[53] E. W. Dijkstra. Self-stabilizing Systems in Spite of Distributed Control. *Commun. ACM*, 17(11):643–644, 1974.

[54] S. Dolev. *Self-stabilization*. MIT Press, Cambridge, MA, USA, 2000. ISBN 0-262-04178-2.

[55] S. Dolev, A. Israeli, and S. Moran. Self-stabilization of Dynamic Systems Assuming only Read/Write Atomicity. *Distributed Computing*, 7:3–16, 1993.

[56] S. Dolev and J. L. Welch. Crash Resilient Communication in Dynamic Networks. *IEEE Transactions on Computers*, 46(1):14–26, 1997.

[57] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff. A Set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Softw.*, 16(1):1–17, 1990.

[58] K. K. Droegemeier, D. Gannon, D. Reed, B. Plale, Alameda J., T. Baltzer, K. Brewster, R. Clark, Domenico B., Graves S., E. Joseph, D. Murray, R. Ramachandran, M. Ramamurthy, Ramakrishnan L., J. A. Rushing, D. Weber, R. Wilhelmson, A. Wilson, M. Xue, and S. Yalda. Service-Oriented Environments for Dynamically Interacting with Mesoscale Weather. *Computing in Science and Engg.*, 7(6):12–29, 2005.

[59] N. Drost, E. Ogston, R. van Nieuwpoort, and H. E. Bal. ARRG: Real-World Gossiping. In *HPDC*, pages 147–158, 2007.

[60] T. Ferrari. Grid Network Services Use Cases from the e-Science Community. In *OGF Grid Final Documents, GFD - I - 122*, 2007.

[61] I. Foster. Globus Toolkit Version 4: Software for Service-Oriented Systems. In *IFIP International Conference on Network and Parallel Computing*, volume 3779 of *LNCS*, pages 2–13. Springer-Verlag, 2005.

[62] I. Foster and C. Kesselman. *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[63] P. Fraigniaud and P. Gauron. The Content-Addressable Network D2B. Technical Report 1349, CNRS - University of Paris Sud, January 2003.

[64] S. Fu, C. Xu, and H. Shen. Random Choices for Churn Resilient Load Balancing in Peer-to-Peer Networks. In *IPDPS'2008*. IEEE, April 2008.

[65] P. Ganesan, B. Yang, and H. Garcia-Molina. One Torus to Rule Them All: Multidimensional Queries in P2P Systems. In *WebDB*, pages 19–24, 2004.

[66] L. Garces-Erice, E. W. Biersack, K. W. Ross, P. A. Felber, and G. Urvoy-Keller. Hierarchical Peer-to-Peer Systems. *Parallel Processing Letters (PPL) Volume 13 N4, 2003*.

[67] L. Garces-Erice, P. A Felber, K. W Ross, and G. Urvoy-Keller. Data Indexing in Peer-to-Peer DHT Networks. In *Proceedings of ICDCS'2004*, Tokyo, Japan, March 2004.

[68] L. Garces-Erice, K. W Ross, E. W Biersack, P. A Felber, and G. Urvoy-Keller. Topology-Centric Look-up Service. In *NGC'03, Munich, Germany*, Sep 2003.

[69] G. Giakkoupis and V. Hadzilacos. A scheme for load balancing in heterogenous distributed hash tables. In *PODC '05: Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, pages 302–311. ACM, 2005.

[70] Gnutella. http://gnutellapro.com, 2008.

[71] B. Godfrey, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica. Load Balancing in Dynamic Structured P2P Systems. In *Proceedings of INFOCOM 2004*, Hong Kong, China, 2004.

[72] B. Godfrey and I. Stoica. Heterogeneity and Load Balance in Distributed Hash Tables. In *INFOCOM*, pages 596–606, 2005.

[73] World Community Grid. http://www.worldcommunitygrid.org, 2008.

[74] P. Grosso, M. Swany, P. Primet, and A. Ceyden. Network Topology Descriptions in Optical Hybrid Networks. In *OGF NML-WG*, 2008.

[75] M. Harren, Hellerstein J., R. Huebsch, B. Loo, S. Shenker, and I. Stoica. Complex Queries in DHT-Based Peer-To-Peer Networks. In *Proceedings of 1st International Workshop on P2P Systems (IPTPS'02)*, Cambridge, MA, USA, March 2002.

[76] T. Herault, P. Lemarinier, O. Peres, L. Pilard, and J. Beauquier. A Model for Large Scale Self-Stabilization. In IEEE, editor, *21th International Parallel and Distributed Processing Symposium, IPDPS 2007*, 2007.

[77] T. Herman and Pirwani I. A Composite Stabilizing Data Structure. In Springer LNCS 2194, editor, *Proceedings of the 5th International Workshop on Self-Stabilizing Systems*, pages 197–182, 2001.

[78] T. Herman and Masuzawa T. A Stabilizing Search Tree with Availability Properties. In IEEE, editor, *Proceedings of the 5th International Symposium on Autonomous Decentralized Systems (ISADS'01)*, pages 398–405, 2001.

[79] T. Herman and Masuzawa T. Available Stabilzing Heaps. *Information Processing Letters*, 77:115–121, 2001.

[80] I. A. Howes, M. C. Smith, and G. S. Good. *Understanding and deploying LDAP directory services*. Macmillian Technical Publishing, 1999.

[81] R. Hsiao and S. De Wang. Jelly: A Dynamic Hierarchical P2P Overlay Network with Load Balance and Locality. In *ICDCS Workshops*, pages 534–540, 2004.

[82] J. Hu, H. Guan, and H. Zhong. A Decentralized Quickest Response Algorithm for Grid Service Discovery. In *InfoScale '07: Proceedings of the 2nd international conference on Scalable information systems*, pages 1–5, ICST, Brussels, Belgium, Belgium, 2007. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[83] A. Iamnitchi and I. Foster. On Death, Taxes, and the Convergence of Peer-to-Peer and Grid Computing. In *2nd International Workshop on Peer-to-Peer Systems (IPTPS'03)*, pages 118–128, 2003.

[84] Sun Microsystems Inc. Java RMI. http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp, 2008.

[85] Java Native Interface. http://java.sun.com/j2se/1.4.2/docs/guide/jni/, 2008.

[86] L. Pearlman N. Miller C. Kesselman J. M. Schopf, I. Raicu, I. Foster, and M. D'Arcy. Monitoring and Discovery in a Web Services Framework: Functionality and Performance of Globus Toolkit MDS4. Technical Report ANL/MCS-P1248-0405, Argonne National Laboratory, 2005.

[87] M. Jain and C. Dovrolis. Pathload: A measurement Tool for End-to-End Available Bandwidth. In *PAM '02: Proceedings of Passive and Active Measurements Workshop*, March 2002.

[88] E. Jeanvoine, C. Morin, and D. Leprince. Vigne: Executing Easily and Efficiently a Wide Range of Distributed Applications in Grids. In *Euro-Par*, pages 394–403, 2007.

[89] F. Kaashoek and D. R. Karger. Koorde: A Simple Degree-optimal Hash Table. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS'03)*, 2003.

[90] D. R. Karger and M. Ruhl. Simple Efficient Load Balancing Algorithms for Peer-to-Peer Systems. In *Proceedings of the 3rd International Workshop on Peer-to-Peer Systems (IPTPS'04)*, pages 131–140, 2004.

[91] K. Kenthapadi and G. S. Manku. Decentralized Algorithms Using Both Local and Random Probes for P2P Load Balancing. In *SPAA'05: Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 135–144. ACM, 2005.

[92] J. Kleinberg. The Small-World Phenomenon: An Algorithmic Perspective. In *Proceedings of the 32nd ACM Symposium on Theory of Computing*, 2000.

[93] C. G. Knight, S. H. E. Knight, N. Massey, T. Aina, C. Christensen, D. J. Frame, J. A. Kettleborough, A. Martin, S. Pascoe, B. Sanderson, D. A. Stainforth, and M. R. Allen. Association of Parameter, Software and Hardware Variation with Large Scale Behavior Across 57,000 Climate Models. *Proceedings of the National Academy of Sciences*, 104:12259–12264, 2007.

[94] J. Laganier and P. Vicat-Blanc Primet. HIPernet: a Decentralized Security Infrastructure for Large Scale Grid Environments. In *6th IEEE/ACM International Conference on Grid Computing (GRID 2005), November 13-14, 2005, Seattle, Washington, USA, Proceedings*, pages 140–147. IEEE, 2005.

[95] J. Ledlie and M. I. Seltzer. Distributed, Secure Load Balancing with Skew, Heterogeneity and Churn. In *INFOCOM*, pages 1419–1430, 2005.

[96] L. Liquori and M. Cosnard. Logical Networks: Towards Foundations for Programmable Overlay Networks and Overlay Computing Systems. In *TGC*, pages 90–107, 2007.

[97] B. Liu, W. Lee, and Lee D. L. Supporting Complex Multi-Dimensional Queries in P2P Systems. In *ICDCS*, pages 155–164, 2005.

[98] B. B. Lowekamp, B. Tierney, L. Cottrell, R. Hughes-Jones, and T. Kielmann. Enabling Network Measurement Portability Through a Hierarchy of Characteristics. In *International workshop Grid2003*, pages 68–75, 2003.

[99] J. Luitjens, B. Worthen, M. Berzins, and T.C. Henderson. Scalable Parallel AMR for the Uintah Multiphysics Code. In D. Bader, editor, *Petascale Computing Algorithms and Applications*. Chapman and Hall/CRC, 2007.

[100] M. Balazinska and H. Balakrishnan and D. Karger. INS/Twine: A Scalable Peer-to-Peer Architecture for Intentional Resource Discovery. In *Proceedings of the International Conference on Pervasive Computing 2002*, 2002.

[101] D. Malkhi, M. Naor, and D. Ratajczak. Viceroy: a Scalable and Dynamic Emulation of the Butterfly. In *PODC*, pages 183–192, 2002.

[102] G. S. Manku. Balanced Binary Trees for ID Management and Load Balance in Distributed Hash Tables. In *PODC*, pages 197–205, 2004.

[103] G. S. Manku, M. Bawa, and P. Raghavan. Symphony: Distributed Hashing in a Small World. In *USITS'03: Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems*, Berkeley, CA, USA, 2003. USENIX Association.

[104] S. Matsuoka, H. Nakada, M. Sato, and S. Sekiguchi. Design Issues of Network Enabled Server Systems for the Grid. http://www.eece.unm.edu/~dbader/grid/WhitePapers/satoshi.pdf, 2000. Open Grid Forum, Advanced Programming Models Working Group whitepaper.

[105] P. Maymounkov and D. Mazieres. Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. In *Proceedings of IPTPS02*, Cambridge, USA, March 2002.

[106] D. R. Morrison. PATRICIA - Practical Algorithm To Retrieve Information Coded in Alphanumeric. *J. ACM*, 15(4):514–534, 1968.

[107] Top 100 multi-project BOINC participants. http://boinc.berkeley.edu/chart_list.php, 2008.

[108] M. Naor and U. Wieder. Novel Architectures for P2P Applications: the Continuous-Discrete Approach. In *SPAA*, pages 50–59, 2003.

[109] Napster. http://www.napster.com, 2008.

[110] M. Nesterenko and A. Arora. Stabilization-Preserving Atomicity Refinement. *Journal of Parallel and Distributed Computing*, 62(5):766–791, 2002.

[111] M. L. Norman, G. L. Bryan, R. Harkness, J. Bordner, D. Reynolds, B. O'Shea, and R. Wagner. Simulating Cosmological Evolution with Enzo. *Petascale Computing: Algorithms and Applications*, 2007.

[112] D. Oppenheimer, J. Albrecht, D. Patterson, and A. Vahdat. Distributed Resource Discovery on PlanetLab with SWORD. In *Proceedings of the ACM/USENIX Workshop on Real, Large Distributed Systems (WORLDS)*, December 2004.

[113] P. Triantafillou and T. Pitoura. Towards a Unifying Framework for Complex Query Processing over Structured Peer-to-Peerp-grid Data Networks. In *Proceedings of the First International Workshop on DataBases, Information Systems and Peer-to-Peer Computing (DBISP2P)*, September 2003.

[114] C. Plaxton, R Rajaraman, and A. Richa. Accessing Nearby Copies of Replicated Objects in a Distribued Environment. In *ACM SPAA*, Newport, USA, June 1997.

[115] R. Prasad, M. Murray, C. Dovrolis, and K. Claffy. Bandwidth Estimation: Metrics, Measurement Techniques, and Tools. *IEEE Network*, 2003.

[116] The P-Grid Project. http://www.p-grid.org/, 2008.

[117] S. Ramabhadran, S. Ratnasamy, J. M. Hellerstein, and S. Shenker. Prefix Hash Tree: An Indexing Data Structure over Distributed Hash Tables. In *Proceedings of the 23rd ACM Symposium on Principles of Distributed Computing*, St. John's, Newfoundland, Canada, July 2004.

[118] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Adressable Network. In *ACM SIGCOMM*, pages 161–172, 2001.

[119] M. Venkateswara Reddy, A. Vijay Srinivas, Tarun Gopinath, and D. Janakiram. Vishwa: A Reconfigurable P2P Middleware for Grid Computations. In *ICPP*, pages 381–390, 2006.

[120] Rhea, S. C. and Geels, D. and Roscoe, T. and Kubiatowicz, J. Handling Churn in a DHT. In *USENIX Annual Technical Conference, General Track*, pages 127–140, 2004.

[121] A. Rowstron and P. Druschel. Pastry: Scalable, Distributed Object Location and Routing for Large-Scale Peer-To-Peer Systems. In *International Conference on Distributed Systems Platforms (Middleware)*, November 2001.

[122] Dolev S. and R. I. Kat. HyperTree for Self-Stabilizing Peer-to-Peer Systems. *Distributed Computing*, 20(5):375–388, 2008.

[123] S. Saroiu, P. Gummadi, and S. Gribble. A Measurement Study of Peer-to-Peer File Sharing Systems. In *Proceedings of Multimedia Computing and Networking*, 2002.

[124] M. K. Satish and Rajendra R. J. GBTK: A Toolkit for Grid Implementation of BLAST. In *HPCASIA '04: Proceedings of the High Performance Computing and Grid in Asia Pacific Region, Seventh International Conference*, pages 378–382, Washington, DC, USA, 2004. IEEE Computer Society.

[125] C. Schmidt and M. Parashar. Squid: Enabling Search in DHT-Based Systems. *Journal of Parallel and Distributed Computing*, 68(7):962–975, July 2008.

[126] M. Schneider. Self-stabilization. *ACM Computing Surveys*, 25:45–67, 1993.

[127] K. Schulten, J. C. Phillips, L. V. Kal, and A. Bhatele. Biomolecular Modeling in the Era of Petascale Computing. *Petascale Computing: Algorithms and Applications*, 2007.

[128] A. Segall. Distributed Network Protocols. *IEEE Transactions on Information Theory*, IT-29:23–35, 1983.

[129] A. Shaker and D. S. Reeves. Self-Stabilizing Structured Ring Topology P2P Systems. In IEEE, editor, *Fifth IEEE International Conference on Peer-to-Peer Computing, P2P 2005*, pages 39–46, 2005.

[130] H. Shen, C. Xu, and G. Chen. Cycloid: A Constant-Degree and Lookup-Efficient P2P Overlay Network. *Performance Evaluation*, 63(3):195–216, 2006.

[131] C. Shirky, editor. *What Is P2P... and What Isn't.* The O'Reilly Network, available at http://www.oreillynet.com/pub/a/p2p/2000/11/24/shirky1-whatisp2p.html, 2008.

[132] B. C. Shu, Y. Ooi, K. Tan, and A. Zhou. Supporting Multi-Dimensional Range Queries in Peer-to-Peer Systems. In *Peer-to-Peer Computing*, pages 173–180, 2005.

[133] J. Siegel. *CORBA: Fundamentals and Programming.* John Wiley & Sons Inc., New York, 1 edition, 1996. ISBN 0-471-12148-7.

[134] D. Spence and T. Harris. XenoSearch: Distributed Resource Discovery in the XenoServer Open Platform. In *Proceedings of HPDC*, pages 216–225, 2003.

[135] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup service for Internet Applications. In *ACM SIGCOMM*, pages 149–160, 2001.

[136] Y. Tanaka, H. Takemiya, H. Nakada, and S. Sekiguchi. Design, Implementation and Performance Evaluation of GridRPC Programming Middleware for a Large-Scale Computational Grid. In *GRID'04: Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing*, pages 298–305, Washington, DC, USA, 2004. IEEE Computer Society.

[137] Y. Tanimura, K. Seymour, E. Caron, A. Amar, H. Nakada, Y. Tanaka, and F. Desprez. *Interoperability Testing for the GridRPC API Specification.* Open Grid Forum, May 2007. OGF Reference: GFD.102.

[138] TeraGrid. http://www.teragrid.org, 2008.

[139] D. Thain, T. Tannenbaum, and M. Livny. Distributed Computing in Practice: the Condor Experience. *Concurrency - Practice and Experience*, 17(2-4):323–356, 2005.

[140] The Distributed ASCI Supercomputer 3. http://www.cs.vu.nl/das3/index.shtml, 2008.

[141] The EGEE project. http://www.eu-egee.org/, 2008.

[142] The gLite middleware. http://glite.web.cern.ch/glite/, 2008.

[143] The Grid NAREGI Program. http://www.naregi.org, 2008.

[144] The Horizon project. http://www.projet-horizon.fr, 2008.

[145] The JXTA project. http://www.jxta.org, 2008.

[146] The KaZaA web site. http://www.kazaa.com, 2008.

[147] The OASIS consortium. http://www.oasis-open.org/, 2008.

[148] The OGSA Working Group. http://forge.ogf.org/sf/projects/ogsa-wg, 2008.

[149] The RENATER Network. http://www.renater.fr, 2008.

[150] B. Traversat, M. Abdelaziz, and E. Pouyoul. A Loosely-Consistent DHT Rendezvous Walker. Technical report, Sun Microsystems, Inc., March 2003.

[151] Web Service Resource Framework. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrf, 2008.

[152] Z. Xu, M. Mahalingam, and M. Karlsson. Turning Heterogeneity into an Advantage in Overlay Routing. In *INFOCOM*, 2003.

[153] Z. Xu, R. Min, and Y. Hu. HIERAS: A DHT Based Hierarchical P2P Routing Algorithm. In *ICPP*, 2003.

[154] Z. Xu and P. K. Srimani. Self-Stabilizing Publish/Subscribe Protocol for P2P Networks. In Springer LNCS 3741, editor, *7th international workshop on Distributed Computing (IWDC 2005)*, pages 129–140, 2005.

[155] Z. Xu and Z. Zhang. Building Low-Maintenance Expressways for P2P Systems. Technical Report HPL-2002-41, Hewlett-Packard Labs, 2002.

[156] A. YarKhan, J. Dongarra, and K. Seymour. GridSolve: The Evolution of Network Enabled Solver. In *Grid-Based Problem Solving Environments: IFIP TC2/WG 2.5 Working Conference on Grid-Based Problem Solving Environments*, pages 215–226, Prescott, USA, July 2006.

[157] C. Zhang, A. Krishnamurthy, and R. Y. Wang. Brushwood: Distributed Trees in Peer-to-Peer Systems. In *IPTPS*, pages 47–57, 2005.

[158] B. Y. Zhao, Y. Duan, L. Huang, A. D. Joseph, and J. D. Kubiatowicz. Brocade: Landmark Routing on Overlay Networks. In *1st International Workshop on Peer-to-Peer Systems (IPTPS'02)*, 2002.

[159] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: A Resilient Global-scale Overlay for Service Deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, January 2004.

[160] Y. Zhu and Y. Hu. Towards Efficient Load Balancing in Structured P2P Systems. In *Proceedings of IPDPS'2004*, April 2004.

[161] Zorilla. http://projects.gforge.cs.vu.nl/ibis/zorilla.html, 2008.

Publications

International Journal Articles

[CDT07] Eddy Caron, Frédéric Desprez, and Cédric Tedeschi. Enhancing Computational Grids with Peer-to-Peer technology for Large Scale Service Discovery. *Journal of Grid Computing*, Volume 5, Number 3, Springer Netherlands, September 2007.

International Conference Articles

[CDPT08] Eddy Caron, Ajoy K. Datta, Franck Petit and Cédric Tedeschi. Self-Stabilization in Tree-Structured Peer-to-Peer Service Discovery Systems. *Twenty-seventh International Symposium on Reliable Distributed Systems (SRDS2008)*. Napoli, Italy. 6-8 October 2008.

[CDT08] Eddy Caron, Frédéric Desprez, and Cédric Tedeschi. Efficiency of Tree-Structured Peer-to-Peer Service Discovery Systems *Fifth International Workshop on Hot Topics in Peer-to-Peer Systems (Hot-P2P 2008)*. Miami, USA. 18 April 2008.

[CDPT07] Eddy Caron, Frédéric Desprez, Franck Petit and Cédric Tedeschi. Snap-stabilizing Prefix Tree for P2P Systems. *Ninth International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2007)*. Paris, France. 12-14 November 2007.

[CDFPT06] Eddy Caron, Frédéric Desprez, Charles Fourdrignier, Franck Petit and Cédric Tedeschi. A Repair Mechanism for Tree-structured Peer-to-peer Systems. *Twelfth International Conference on High Performance Computing (HiPC 2006)*. Bangalore, India. 18-20 December 2006.

[CDT06] Eddy Caron, Frédéric Desprez, and Cédric Tedeschi. A Dynamic Prefix Tree for the Service Discovery Within Large Scale Grids. *Sixth IEEE International Conference on Peer-to-Peer Computing (P2P2006)*. Cambridge, UK. September 6-8, 2006.

[CDPT05] Eddy Caron, Frédéric Desprez, Franck Petit, and Cédric Tedeschi. A Peer-to-Peer Extension of Network-Enabled Server Systems. *First IEEE International Conference on e-Science and Grid Computing (e-Science 2005)*. Melbourne, Australia. 5-8 December, 2005.

National Conference Articles

[Ted08] Cédric Tedeschi. Arbre de préfixes auto-stable pour les systèmes pair-à-pair *18^{e} Rencontres francophones du paralllisme (RenPar'18)*. Fribourg. 11-13 fvrier, 2008.

[Ted06] Cédric Tedeschi. Dcouverte de Services Flexible pour les Grilles de Calcul Dynamiques Large chelle. *17e Rencontres francophones du paralllisme (RenPar'17)*. Perpignan. 4-6 octobre, 2006.

[CFPT06] Eddy Caron, Charles Fourdrignier, Franck Petit, Cédric Tedeschi. Mcanisme de rparations pour un systme pair--pair de dcouverte de services. *17e Rencontres francophones du paralllisme (RenPar'17)*. Perpignan. 4-6 octobre, 2006.